Model-Driven Security: Exemplified for Information Flow Properties and Policies

Doctoral Dissertation by

Fredrik Seehusen

Submitted to the Faculty of Mathematics and Natural Sciences at the University of Oslo in partial fulfillment of the requirements for the degree ph.d. in Computer Science

January 2009
Abstract

As computerized systems have become an important part of society and its infrastructure, the need for secure systems has become increasingly evident. Today, particularly with the emergence of the Internet, the need for security concerns nearly every user of computerized systems, be it private users, industrial users, or government users.

Many security experts believe that security should be taken into account throughout the whole system development process. One way of achieving this, is to integrate security into the Model-Driven Architecture (MDA) framework. MDA aims to raise the level of abstraction of the programming environment by supporting the specification of abstract models that can be transformed down to models of lower levels of abstraction (e.g., code). The integration of security into MDA (called MDS for Model-Driven Security) enables (1) security requirements to be formulated and verified at high levels of abstraction in early phases of system development, and (2) security analysis results to be maintained by transformations to the lower levels of abstraction.

We propose two methods to MDS. The first method defines a stepwise and modular development process in which abstract UML inspired state machines are shown to be in adherence with secure information flow properties, and transformed or refined towards the implementation level in such a way that adherence to the information flow properties is preserved. The second method to MDS defines a development process in which prohibition policies are specified by high level UML sequence diagrams, and transformed into lower level policies that are automatically enforced by runtime monitoring mechanisms.

Both methods are supported by a formal foundation that enables rigorous security analysis by precisely defining the key aspects of MDS.
Acknowledgements

First of all, I would like to thank my advisor Ketil Stølen, for everything that I’ve learned from him, for his for his excellent advice and support, for the countless scientific discussions we had, and for always being there to help not matter what the time or date. I would also like to thank my parents, Erik Seehusen and Turid Seehusen, for all their love and support.

I was fortunate to present my work to guest scientists and principal researchers of the SARDAS project and the ENFORCE project. I would like to thank the following for their valuable feedback: Manfred Broy, Øystein Haugen, Cliff Jones, Fabio Massacci, and Thomas Weigert.

Thanks to the people of the Cooperative and Trusted Systems department at SINTEF Information and Communication Technologies, and its research director Bjørn Skjellaug, for providing a good work environment.

At one point or another, during the work on this thesis, I’ve shared office with Gyrd Brændeland, Ida Hogganvik, Mass Soldal Lund, Tom Lysemose, and Bjørnar Solhaug. Thanks to all of you for being good colleagues, in particular Mass Soldal Lund whom I’ve shared office with for the longest period of time, who despite playing solitaire all day, seemed to get more work done than me anyway. Thanks to the other colleagues in my group, Heidi Dahl and Atle Refsdal, for being good colleagues also.

Finally, I would like to thank all my closest friends for being good friends and for good or bad, putting my mind off work: Bjørn Håvar Falck Andersen, Frode Engelstad, Ole-Andre Karlsson, Jo Masdalen, Tor-Even Ramberg, Kim Sevallsen, Ole Morten Skryseth, Øystein Vestgården, and Monica Wulff-Nilsen.
List of original publications


The publications 1-6 are available as Chapters 9 - 14 in Part II. In the case of publications 1, 2, and 6, we have included the full technical reports which are extended, slightly revised versions of the published papers.
List of original publications
Contents

Abstract iii
Acknowledgements v
List of original publications vii
Contents ix
List of figures xiii

I Overview 1

1 Introduction 3
1.1 Our work ......................................................... 4
1.1.1 A formal foundation for model-driven information flow security 5
1.1.2 A method for model-driven information flow security ........ 5
1.1.3 A method for model-driven policy specification ............... 5
1.2 Structure of the thesis ........................................... 5

2 Problem characterization 7
2.1 Model-driven architecture (MDA) ............................... 7
2.2 Security ........................................................... 9
2.3 Model-driven security (MDS) .................................... 10
2.4 Requirements to the artifacts .................................... 10
2.4.1 The formal foundation for model-driven information flow security 10
2.4.2 The method for model-driven information flow security ....... 11
2.4.3 The method for model-driven policy specification ............ 12

3 State of the art 13
3.1 The formal foundation for model-driven information flow security . 13
3.1.1 Security frameworks ........................................... 14
3.1.2 Secure information flow and refinement of underspecification .. 15
3.1.3 Secure information flow and data refinement ................... 16
3.1.4 Secure information flow and composition ........................ 17
3.2 The method for model-driven information flow security ........... 18
3.3 The method for model-driven policy specification .................. 19
4 Research method
4.1 A research method for technology research ........................................ 23
4.2 How we have applied the research method ........................................... 24
   4.2.1 The formal foundation for model-driven information flow security .... 25
   4.2.2 The method for model-driven information flow security ................. 26
   4.2.3 The method for model-driven policy specification ......................... 27

5 Achievements: the overall picture ........................................................ 29
5.1 Overall picture ...................................................................................... 29
5.2 The formal foundation for model-driven information flow security ....... 30
   5.2.1 The security framework ............................................................... 31
   5.2.2 Preservation of security under transformation and composition .... 31
5.3 The method for model-driven information flow security .................. 32
   5.3.1 Tool support ............................................................................. 34
5.4 The method for model-driven policy specification ......................... 35
   5.4.1 Tool support ............................................................................. 36

6 Overview of research papers ................................................................. 41
6.1 Paper 1: Information Flow Property Preserving Transformation of UML
   Interaction Diagrams ........................................................................... 41
6.2 Paper 2: Maintaining Information Flow Security under Refinement and
   Transformation .................................................................................. 41
6.4 Paper 4: Adherence Preserving Refinement of Trace-set Properties in
   STAIRS: Exemplified for Information Flow Properties and Policies .... 42
6.5 Paper 5: A Method for Model-Driven Information Flow Security ....... 43
6.6 Paper 6: A Transformational Approach to Facilitate Monitoring of High
   Level Policies .................................................................................... 43

7 Discussion .............................................................................................. 45
7.1 The formal foundation for model driven information flow security .... 45
7.2 The method for model driven information flow security .................... 46
7.3 The method for model driven policy specification ............................. 47
7.4 Design decisions .................................................................................. 47
   7.4.1 The formal foundation for model-driven information flow security 48
   7.4.2 The method for model-driven information flow security ............. 48
   7.4.3 The method for model-driven policy specification .................... 49
7.5 Related work ....................................................................................... 50
   7.5.1 Thomas Santen, Maritta Heisel and Andreas Pfitzmann: Confidentiality
       Preserving Refinement .................................................................. 50
   7.5.2 David Basin, Jürgen Doser, and Torsten Lodderstedt: SecureUML .. 51
   7.5.3 Jan Jürjens: UMLSec ................................................................. 52

8 Conclusion ............................................................................................. 55
8.1 What has been Achieved ..................................................................... 55
8.2 Directions for Future Work ................................................................. 56

Bibliography ............................................................................................ 57
II Research Papers

9 Paper 1: Information Flow Property Preserving Transformation of UML Interaction Diagrams 69

10 Paper 2: Maintaining Information Flow Security under Refinement and Transformation 103


List of Figures

2.1 Traditional development cycle (left) and MDA development cycle (right) 8
4.1 Method for technology research ............................................. 24
4.2 Research method for development of artifacts (illustration from [100]) 26
5.1 Transformation of security requirement (LHS) and transformation of system specification (RHS) .................................................. 30
5.2 Combination of the two methods ............................................. 30
5.3 Use cases of the MDSTool .................................................. 33
5.4 State machine diagram editor combined with the MDSTool ............... 35
5.5 Illustration of the output of the three method steps ......................... 36
5.6 Example of a sequence diagram specified in the SeDi editor ............... 37
5.7 Illustration of the TTool setup ............................................. 38
5.8 Graphical user interface of the TTool ..................................... 38
5.9 Steps need to create a new transformation .................................. 39
5.10 Graphical user interface of the create new transformation window ...... 40
Part I

Overview
Chapter 1

Introduction

The importance of security in computerized systems is not new. In 1988, approximately 5000 computers throughout the Internet were rendered unusable within 4 hours by a program called a worm [16, 47]. Since then, security incidents have grown both in number and severity. Today, security incidents occur on a daily basis within most companies, and the systems that surround us are vulnerable to attacks [68]. In CSI/FBI Computer Crime and Security Survey for 2005 [21], 74% of the companies reported security incidents.

Despite the importance of security, careful engineering of security into overall design is often neglected and security features are typically built into an application in an ad-hoc manner or are only integrated during the final phases of system development [77]. According to [77], there are three reasons for this. First, security is a complex attribute that is affected by nearly every component of a system, and its integration into the software development process is not well understood. Second, there is a lack of methods and tools that support security engineering. Third, software developers are generally not security experts, and the integration of security into a system is difficult and errors often arise due to lack of experience of the individual developers.

Although security engineering is not a well developed field, there is a large number of methods and tools that support the development of functional aspects of software systems. One state of the art development framework is the Model-Driven Architecture (MDA) [96] which is advocated by the Object Management Group (OMG). MDA aims to raise the level of abstraction of the programming environment by supporting (1) a model-driven development process, (2) a clear separation of abstract, platform independent models (PIMs) and refined, platform specific models (PSMs), and (3) transformations between PIMs and PSMs.

Integrating security into the MDA framework is interesting for several reasons:

• It reduces the need of ad hoc integration of security mechanisms after the system has been implemented.

• Security analysis is more feasible at the abstract (PIM) level because the concrete (PSM) level may include too much detail to make analysis practical. Also, it is commonly known that the earlier an error is discovered, the easier and cheaper it is to fix [78].

• Abstract specifications are in general more platform independent than concrete specifications. This means that analysis results at the abstract level are more reusable than analysis results at the concrete level.
The specialization of MDA to security is called Model-Driven Security (MDS) [11]. MDA methods in general are often practical in the sense that their associated languages and tools are often widely used in the software industry. However, they are usually not sufficiently precise to exploit the full potential of MDS. In order to make full use of abstract analysis we need to know that the transformations preserve the validity of analysis results. To guarantee this, we need a rigorous underlying semantics characterizing specifications as well as transformations.

It has long recognized that the notions of refinement and composition are important ingredients of software development [126]. Refinement supports a stepwise development process in which an abstract specification is gradually made more concrete. Composition supports a modular development process in which a (composite) system specification is developed, not as a monolithic entity, but by putting together, or composing, other (basic) specifications.

It is our belief that most software development methods will benefit from taking the notions of refinement and composition into account. Moreover, any MDS method should support the preservation of security under refinement and composition. Otherwise the security would have to be reestablished for each refinement step or for each composition.

In summary, our goal is to develop artifacts that support:

- a stepwise modular development process in which transformations play an important role;
- a precise and sound foundation for rigorous security analysis;
- preservation of security under refinement, transformation, and composition.

1.1 Our work

The notion of security considered in our work is in the form of secure information flow properties [40, 81, 122] and properties that can be described by so-called prohibition security policies [110]. Secure information flow properties (referred to as security properties for short) provide a way of specifying requirements on indirect flow of information between different security domains. In contrast to direct information flow which occurs as a result of unauthorized access to data, indirect information flow cannot in general be prevented by traditional access control models. For instance, a Trojan horse program might be secure w.r.t. a traditional access control model, but nevertheless signal confidential information to an attacker via a covert channel.

Security policies are rules governing the choices in the behavior of a system [118]. A prohibition policy is a description of behavior which must not be allowed by the system to which the policy applies. This kind of policies are of the same form as so-called safety properties [6], and can therefore be enforced at runtime by monitoring the execution of the system to which the policy applies.

The main contributions of this thesis are the following artifacts: a formal foundation for model-driven information flow security, a method for model-driven information flow security, and a method for model-driven policy specification. We give a brief overview of each of these artifacts.
1.1.1 A formal foundation for model-driven information flow security

The formal foundation supports the stepwise and modular development of secure software systems. The foundation provides a formal characterization of information flow security properties and their relationship to the notions of refinement, transformation from one level of abstraction to another, and composition. In particular, the foundation defines (1) a general security framework for specifying secure information flow properties that are preserved under refinement, (2) general notions of transformation and composition, and (3) conditions under which secure information flow properties are preserved under transformation and composition.

1.1.2 A method for model-driven information flow security

The method for model-driven information flow security defines a stepwise and modular development process that is supported by the formal foundation for model-driven information flow security. Initially, the user of the method (i.e., a software developer) specifies the system architecture and selects a set of security requirements (in the form of secure information flow properties) that the system must adhere to. The user then specifies each component of the system architecture using UML inspired state machines, and refines these (abstract) components by removing alternative design decisions until a sufficiently concrete level of abstraction is reached. It is assumed that the state machines may reference events that are already provided in a predefined event library. Thus, in the final step of the method, the state machine specifications containing event references are transformed into (more concrete) state machines by replacing the event references by their definitions in the event library.

It is shown how the formal foundation (described in Sect. 1.1.1) can be used to reason about the preservation of security under refinement and transformation.

1.1.3 A method for model-driven policy specification

The method for model-driven policy specification enables the user of the method (i.e., a software developer) to specify high level UML sequence diagram policies that can be enforced by runtime monitoring mechanisms. The method has three main steps. In step I, the user formalizes a set of policies using UML sequence diagrams. In step II, the user selects a set of transformation rules (expressed in UML sequence diagrams) from a transformation library, and applies these to the sequence diagram policies to obtain an intermediate low level policy. In step III, the intermediate low level policy is transformed into a UML inspired state machine that governs the behavior of a runtime monitoring mechanism.

A formal characterization of what it means that a system adheres to a sequence diagram and a state machine policy as well as a formal definition of the transformation from sequence diagrams to state machines are provided.

1.2 Structure of the thesis

This thesis is based on a collection of 6 research papers and split into two parts. Part I provides the context and an overall view of the work, while Part II contains the research...
papers. Part I is organized into the following chapters:

**Chapter 1 - Introduction** presents the main goal of this thesis, the research domain, the contribution, and the structure of this thesis.

**Chapter 2 - Problem characterization** provides background material and argues why the main goal of this thesis is desirable. In addition, success criteria that must be fulfilled by our artifacts to reach our goal successfully are presented.

**Chapter 3 - State of the art** introduces the state of the art literature related to the three artifacts developed in this thesis.

**Chapter 4 - Research method** discusses a method for technology research and how this method has been applied in the development of this thesis.

**Chapter 5 - Achievements: the overall picture** provides an overview of the three invented artifacts of this thesis.

**Chapter 6 - Overview of research papers** provides publication details of each research paper of this thesis.

**Chapter 7 - Discussion** argues that our artifacts satisfy the success criteria stated in Chapter 2. In addition, important design decisions are discussed as well as the related research which is most relevant to our thesis.

**Chapter 8 - Conclusion** presents conclusions and possible areas of future work.

Each research paper in Part II can be read independently of each other. However, we recommend that they be read in the order they appear in the thesis.
Chapter 2

Problem characterization

As discussed in Chapter 1, our goal is to develop stepwise and modular approaches to MDS that allow for precise and rigorous analysis. In this chapter, we present some background material and motivate in more detail why this goal is desirable. Furthermore, we decompose and refine this goal into a set of success criteria that we would like our artifacts to fulfill.

2.1 Model-driven architecture (MDA)

Model-driven architecture (MDA) is a framework for software development defined by the Object Management Group (OMG) [44, 96]. MDA builds on and brings together several important advances in computer science such as object-oriented development, component based development, design patterns, distributed computing and declarative specification techniques [38]. In essence, MDA aims to raise the level of abstraction of the programming environment by supporting

- a model-driven development process;
- a clear separation of abstract, platform independent models (PIMs) and refined, platform specific models (PSMs);
- transformations between PIMs and PSMs.

Model-driven development is software development with models as the artifacts driving the development, as opposed to, e.g., software development with code as the driving artifacts [8, 27, 80]. However, as illustrated in Fig. 2.1 (illustration taken from [69]), the main difference between traditional software development and MDA software development is the relationship of the artifacts that are produced at the analysis and design time of the software development cycle [69]. In a traditional development process, it is often the case that the artifacts produced at the first three phases of the development cycle are in the form of text or diagrams. However, most of the artifacts from these phases are just paper and nothing more. When the coding starts, the artifacts produced in these phases loose their value as their relationship to the code is lost. Instead of being an exact specification of the code, the diagrams usually become more or less unrelated pictures [69].

In the MDA lifecycle, the artifacts that are produced at analysis and design time are models that can be understood by computers and transformed to code. In particular, MDA distinguishes between three kinds of models: Platform Independent Models
Figure 2.1: Traditional development cycle (left) and MDA development cycle (right)

(PIMs), Platform Specific Models (PSMs), and code. The PIM is a model with a high level of abstraction that is independent of any implementation technology. A PSM is tailored to constructs that are available on a particular platform. The code represents the final description of the system.

The main advantage of MDA software development over traditional development is that the artifacts produced at analysis and design time become more valuable, particularly in later stages of software development, and the effort in deriving code from these artifacts becomes less time consuming. MDA is believed to improve productivity, portability, interoperability, and maintenance and documentation [69].

A central term of the MDA framework is the notion of model. According to [69], “A model is a description of (part of) a system written in a well-defined language.”

The term model is often used to refer to abstractions above code. Although models often have a visual notation (concrete syntax), this is often not understood as being a defining characteristic of models [22]. Note that in this thesis, we will often use the term specification instead of model.

The de facto modeling language of MDA is the Unified Modeling Language (UML) [46]. UML was developed in an effort to simplify and consolidate a large number of object-oriented development methods [102]. UML is a general-purpose visual modeling language that is used to specify, visualize, construct, and document the artifacts of a software system.

Model transformations play an essential role in the MDA framework. According to [69], “a model transformation is the automatic generation of a target model from a source model, according to a transformation definition.”

A transformation definition can be executed by a transformation tool in order to transform PIMs to PSMs, or PSMs to code. For short, we will often use the term
transformation to denote a transformation definition when it is clear from the context what is meant.

Model transformation is closely related to program transformation. There is no big conceptual difference between the two notions. Their differences occur in the mindsets and traditions of their respective transformation communities [22]. While model transformation is a relatively new field whose transformation definitions usually adopt an object-oriented approach, program transformation is usually based on more mathematically oriented concepts such as term rewriting [9], attribute grammars [70], and functional programming [56].

There are quite a number of model-based transformation approaches. A classification and a survey of these approaches can be found in [22].

The notion of transformation is closely related to the notion of refinement [25, 54]. That is, refinement is the (possibly manual) process of bringing an abstract specification closer to an implementation, while a transformation may be seen as a special case of refinement where this process is automated. Refinement is one of the most important activities during design [126]. A precise characterization of refinement provides a foundation for a systematic approach to software development. This question has therefore received considerable attention in the literature (see e.g., [54, 63]). We believe that any MDA method will benefit from taking the notion of refinement into account. The same holds for the notion of composition. That is, history has shown that composition is a useful way of coping with the difficulty of reasoning about large systems [64]. In a compositional approach, the overall system is divided into components that can be developed independently. Analysis results for the overall system should follow from analysis of its components as this greatly increases the feasibility of analysis.

2.2 Security

As computerized systems have become an important part of society and its infrastructure, the need for secure systems has become increasingly evident. Today, particularly with the emergence of the Internet, the need for security concerns nearly every user of computerized systems, be it private users, industrial users, or government users.

According to [60], security is defined as “the preservation of confidentiality, integrity, and availability”. Confidentiality is about the concealment of information or resources [16]. Integrity refers to the trustworthiness of data resources, and it is usually phrased in terms of preventing unauthorized change [16]. There are two kinds of integrity: data integrity (the content of the information) and origin integrity (the source of data, often called authentication). Availability refers to the ability to use the information or resource desired [16]. Threats to availability include so-called denial-of-service attacks.

The first formal work in the security area was motivated by the military’s attempt to enforce a “need to know” principle [16]. Some of the well-know models of security include the access control matrix [42, 49, 73], the schematic protection model [104, 105], the take-grant protection model [76], and information flow security models [40, 122].

The distinction between a policy and a mechanism is crucial to the study of security. A security policy is a set of rules governing the choices in the behavior of a system [118] whereas a security mechanism is a method, tool, or procedure for enforcing a security policy [16]. The most well-know mechanisms enforcing security are access control mechanisms, cryptography mechanisms, firewalls, and virus scanners.
2.3 Model-driven security (MDS)

Although security has received much attention in the research community, not many would agree that security has improved over the years. There are many reasons for this. First, computerized systems have become more complex and more connected. Second, developing security systems is difficult. Security is a property that is affected by nearly every component of a system. This makes it hard to pinpoint security vulnerabilities and protect against specific security threats. Third, there is a lack of tools that support the development of secure systems. Fourth, many software developers lack the knowledge required to develop secure systems. Fifth, system functionality is usually given priority over non-functional properties such as security. Under the pressure of tight deadlines, security may not be taken into consideration at all.

Model-driven security (i.e., the integration of security into MDA) offers a solution to many of these problems. MDS enables (1) security requirements to be formulated and verified at high levels of abstraction in early phases of system development, and (2) security analysis results to be maintained by transformations to the lower levels of abstraction. Thus MDS enables security to be taken into account from start to finish during system development.

Approaches to MDA or related technologies are often practical in the sense that they are often tool supported and widely used in the software industry. However, they often not sufficiently precise to facilitate the full benefits of MDA. In order to make full use of abstract analysis, we need to know that transformations preserve the validity of the analysis results. To guarantee this we need a rigorous underlying semantics characterizing the exact meaning of specifications, security, and transformations.

Intuitively, one might expect that security is a more intangible property than functional properties like safety and liveness properties. This intuition is confirmed by the work of Jacob [61] in which it is stated that preservation of security properties under refinement is fundamentally harder than preservation of so-called safety or liveness properties.

2.4 Requirements to the artifacts

In this section, we state a number of requirements that the artifacts presented in Sect. 1.1 should fulfill.

2.4.1 The formal foundation for model-driven information flow security

The overall success criterion to the formal foundation is that it should precisely define the key notions of a model-driven information flow security development process. This criterion is in the following broken down into five criteria.

Success criterion 1: The foundation should precisely define what is meant by a specification.

A specification is a central notion of any software development method. Therefore, the formal foundation should precisely and unambiguously define what is meant by a specification. The definition should as far as possible be independent of any notation or technology.
Success criterion 2 The foundation should precisely define what is meant by information flow security.

Since the notion of security we consider is in the form of secure information flow properties, the foundation should precisely define what it means that a specification is secure w.r.t. an information flow security property. It is desirable that this is defined in a general way such that attention is not restricted to particular secure information flow properties.

Success criterion 3 The foundation should precisely define what it means that security is preserved under refinement.

Refinement has proven to be a useful notion in software development. However, there is little point in analyzing abstract specifications if the results of the analysis are not preserved down to the final implementation. Therefore, the method should handle preservation of security under refinement.

Success criterion 4 The foundation should precisely define what it means that security is preserved under transformation.

This criterion is similar to the previous criterion. However, refinement and transformation are two different notions; a refinement is a relation between specifications, whereas a transformation is an executable function that constructs a specification from another specification.

Success criterion 5 The foundation should precisely define what it means that security is preserved under composition.

In a compositional development process, the system is not developed as a monolithic entity. Instead it is constructed by putting together or composing system parts. For reasons of efficiency and reusability it is of interest to analyze each part that a system consists of separately. This is most useful when security is preserved under composition, i.e., when the composed system is secure whenever its parts are secure. Therefore the method should characterize conditions under which security is preserved by composition.

2.4.2 The method for model-driven information flow security

Success criterion 6 The method should enable security analysis at a high level of abstraction.

The fulfillment of this criterion should ensure that security is taken into account in the early phases of software development, e.g., design time.

Success criterion 7 The method should support preservation of security under refinement and transformation.

The fulfillment of this criterion ensures that security analysis results already established at an abstract level do not have to be reestablished for each refinement or transformation step. Adding details to a specification under refinement or transformation may of course require some additional analysis. However, it should not be necessary to recheck the adherence relationship already established at the more abstract level.
Success criterion 8  *Be easy to understand and employ by the users of the method (which we assume are software developers).*

This criterion should ensure that the method is usable for software developers that do not have a theoretical background. The criterion entails that the method must “hide” technical details of the formal foundation which are not necessary for the employment of the method.

Success criterion 9  *The method should be supported by tool.*

Without tool support, it is unlikely that the method will ever be used in practice in an industrial setting. The tool should ideally automate as much of the method as possible, i.e., security analysis and transformation. It should also support the specification of security requirements, software systems, and transformations.

### 2.4.3 The method for model-driven policy specification

Success criterion 10  *The method should support the specification of policies at a high level of abstraction.*

Since our goal is to develop approaches to MDS in which the distinction of abstract and concrete specifications is essential, the method should support the specification of policies at a high level of abstraction.

Success criterion 11  *The method should support transformation of high level policies to low level policies.*

In general, it is desirable that the method addresses the problem of how to derive low level policies from high level policies. One of the best ways of solving this problem, we believe, is to support a transformation from the high level to the low level.

Success criterion 12  *The method should facilitate automatic enforcement of low level policies.*

Since there are existing mechanisms available with which to enforce adherence of a system to a policy, it is clearly desirable that the specified policies can be enforced by such a mechanism.

Success criterion 13  *The method should be easy to understand and employ by the users of the method (which we assume are software developers).*

This criterion is the same in Sect. 2.4.2.

Success criterion 14  *The method should be supported by a tool.*

Tool support is a necessary criterion if the method should be applied in practice. The tool should support the specification of policies, transformation of high to low level policies, and the enforcement of low level policies.
Chapter 3
State of the art

In this chapter, we give a general overview of state of the art literature of relevance for the three artifacts developed in this thesis. For a more detailed discussion on the relationship of our artifacts to the literature, the reader is referred to Sect. 7.5 and to the related work sections in the attached papers.

3.1 The formal foundation for model-driven information flow security

It is useful to distinguish between two kinds of insecurity \[85\]. The first kind of insecurity allows an attacker to bypass normal access control to directly receive classified information. The Bell-LaPadula model \[14\] and the Biba model \[15\] are intended to prevent this kind of insecurity. E.g., for a system to be secure in the sense of Bell-LaPadula, every possible sequence of the system state transitions must result in a secure state; i.e., one in which no user has access to classified data unless he is authorized to have that access. The second kind of insecure system prevents direct unauthorized access, i.e., it is secure in the sense of Bell-LaPadula, but it allows for covert channels. A covert channel is an indirect communication path between users. For the second kind of insecure systems, it is often necessary for the attacker to have a Trojan horse program which is privileged to see classified data and can signal this information to the attacker. The prevention of this kind of insecurity is the motivation of secure information flow requirements\(^1\).

Information flow security was first introduced in \[122\] as a generalization of the notion of non-interference \[40, 41\] (which applies to deterministic systems) to non-deterministic systems. Given a system with one public (i.e., low level) and one confidential (i.e., high level) interface, non-interference requires that confidential inputs never affect public outputs of the system. If this property holds, we can conclude that no information flow is ever possible from high to low level. Information flow properties in general make use of a more general notion of low level observations and confidential behavior, but the underlying idea is much the same, namely to prevent the low level observations from being influenced by confidential behavior.

Many information flow properties have been proposed in the literature in various semantic models such as trace-based models \[37, 81, 84, 87, 88, 132\], state-based automation models \[40, 90, 95, 127, 129\], and process algebraic models \[34, 94, 101, 111\]. See \[89\] for a

\(^1\)Sometimes called non-interference or possibilistic security requirements.
comparison of security properties across the semantic frameworks. Examples of secure
information flow properties are non-inference [94], generalized non-inference [87], gen-
eralized non-interference [84], restrictiveness [84], the perfect security property [132],
and separability [87].

3.1.1 Security frameworks

Various security frameworks have been proposed in which security properties can be
formulated and compared [7, 17, 36, 81, 87, 88, 132]. We give a brief review of these
frameworks.

In [87, 88], McLean considers specifications that are interpreted as sequences of
states called traces. In McLean’s framework, all security properties are closure condi-
tions\(^2\) under so-called selective interleaving functions. A selective interleaving function
takes two traces as input and produces the trace obtained by selecting state-objects
from its arguments. According to [81], the expressiveness of McLean’s framework is
limited in that it cannot capture inductive definitions which are required for properties
such as the perfect security property [132].

Zakinthinos and Lee [132] consider specifications expressed as traces of events. Their
framework is based on so-called low level equivalence sets, i.e., the set of all traces
containing the same low level observation. The framework provides little structure
in defining security properties, and according to [81], the framework is not expressive
enough to capture properties like separability and the perfect security property.

In [81], Mantel presents the so-called MAKS (Modular Assembly Kit for Security
properties) framework based on traces of events. In Mantel’s framework, each security
property is specified as a conjunction of so-called basic security predicates. A basic
security predicate is schema that is parameterized by a restriction and a closure con-
dition and requires that for each trace \(t\) of a system that satisfies the restriction, there
must be a trace \(u\) with the same low level observation as \(t\) and that satisfies the closure
condition. The idea is that the presence of \(u\) prevents the low level user from asserting
that high level behavior has or has not occurred. Mantel’s framework provides more
structure than the framework of Zakinthions and Lee. At the same time it is sufficiently
expressive to capture security properties such as the perfect security property.

Alur et. al. [7] considers specifications interpreted as traces of states and events.
The framework is based on a so-called inferable property function that given a trace
\(t\), a property \(\alpha\), and an equivalence relation \(\equiv\) over traces, yields the knowledge of
the observer about the property \(\equiv\) after trace \(t\) has been executed. They say that
a property \(\alpha\) is secret w.r.t. a given equivalence relation \(\equiv\), if for every trace \(t\) of a
system, the low level observer cannot assert that the property \(\alpha\) holds or not holds
when \(t\) is executed. The framework seems to be sufficiently general to capture the
perfect security property, but the it is less modular than Mantel’s framework.

The framework of Bossi et. al. [17] is defined in a process algebraic setting. It is
based on a generalization of a so-called unwinding condition introduced in [41] to prove
systems secure. Intuitively, the generalized unwinding condition states that if there is
a high level transition from one state \(S\) to another state \(S'\), then there should be a
“simulated” transition from \(S\) to another state \(S''\) which is low level equivalent to \(S'\).
If this holds, then the low level user cannot assert that the system has moved into state

\(^2\)A set \(A\) is closed under a function \(f\) iff \(s \in A\) implies that \(f(s) \in A\).
3.1 The formal foundation for model-driven information flow security

Like the framework of Alur et. al., the framework of Bossi et. al. is quite general, but it seems to be less modular than Mantel’s framework.

Focardi and Martinelli [36] present a uniform approach for the definition of security properties. Their model is based on the so-called Cryptographic Security Process Algebra (CryptoSPA) language. CryptoSPA is an extension of CCS [91] with primitives for manipulating messages. Focardi and Martinelli propose a general schema, called the Generalized Non Deducibility on Compositions (GNDC), for the definition of security properties. The main idea is that a system is secure iff for every possible environment, the composition of the system and the environment is secure. The authors show how the GNDC schema can be instantiated to obtain standard information flow properties as well as authentication and non-repudiation properties. The framework is therefore quite general, but like the generalized unwinding condition, it seems less modular than Mantel’s framework.

3.1.2 Secure information flow and refinement of underspecification

In design specifications, nondeterminism often represents underspecification, i.e., design alternatives that are equivalent in the sense that it does not matter which one of them is implemented. Preserving information flow properties under refinement of underspecification has proven to be problematic. To our knowledge, Jacob [61] was the first to consider this problem. The notion of refinement he considers is based on the so-called safety ordering [55]. According to this notion, a specification $S'$ (interpreted as a set of traces) is a refinement of a specification $S$ if $S'$ is a subset of $S$. This notion of refinement is known to preserve safety and liveness properties [6], but Jacob shows that this is not the case for security properties.

Since Jacob’s initial investigation, a large number of papers have addressed the relationship of information flow security and refinement. These can be classified into two categories: those that propose conditions under which a given notion of refinement preserves standard security properties [7, 17, 18, 52, 82, 106–108], and those that reformulate standard security properties in such a way that they are preserved under refinement, e.g., by closing the security properties under refinement [67, 78, 79, 101].

In the following, we give a brief presentation of the above citations.

Mantel [82] considers the same notion of refinement as Jacob, i.e., trace-set inclusion. To overcome the refinement problem, he presents a collection of so-called refinement operators that characterize secure refinements. According to [52], the refinement operators may lead to concrete specifications that are practically hard to implement, because the changes in the refinement they induce are hard to predict and may not be easy to realize in an implementation.

Bossi et. al. [18] investigate security and refinement in a process algebraic setting. The security properties addressed are those that can be expressed as instances of the generalized unwinding condition. Refinement is defined such that a process $E$ is a refinement of a process $F$ if $E$ is simulated by $F$. This is a stronger notion than the trace-set inclusion refinement considered by Mantel and Jacob, but it is not, in general, security preserving. Therefore, a general condition under which security is preserved is proposed. It is shown that some of the results of Mantel can be obtained as instances of this condition.

In [7], security properties are defined over traces of states and events. A notion of
secrecy-preserving refinement is presented. The idea is that specification $S_c$ is a secrecy preserving refinement of specification $S_a$ if for each trace $t_c$ of $S_c$, there is an equivalent trace $t_a$ of $S_a$ such that the low level observer can deduce less about the properties of interest when observing $S_c$ execution $t_c$ than observing $S_a$ executing $t_a$.

Santen et al. [52, 106–108] consider CSP (Communicating Sequential Processes [55]) specifications extended with a probabilistic choice operator. The notion of refinement investigated is CSP refinement (trace refinement, failure refinement, and failure-divergence refinement) modulo a so-called retrieve relation that maps concrete data to abstract data. A notion of confidentiality preserving refinement is defined. The underlying idea is similar to [7, 18], i.e., that a low level observer must not gain more information at the concrete level than at the abstract level. The papers discussed so far propose conditions under which refinements are secure. Another solution to the problem is to formulate security properties in such a way that they are preserved under refinement [67, 78, 79, 101]. Lowe argues [78, 79] that this idea not only ensures preservation under refinement, but that it also leads to more intuitive notions of security. In [79], Lowe therefore presents a new security property which is closed under refinement. Unlike Jacob or Mantel, he is able to close a property under refinement without making the property unreasonably strong because the notion of refinement investigated by Lowe is less liberal than the one considered by Jacob or Mantel. A similar line of reasoning is followed by Jürgens in [67], where a secure information flow property is formalized such that each behavior refinement to a deterministic system satisfies the property.

Roscoe [101] takes a more drastic approach. He argues that a system should only be considered secure if it is deterministic from a low level point of view. The idea is that the only way information can leak from the high level to the low level is via the process behaving differently towards the low level user depending on what the high level user has done. These kind of security properties are closed under CSP notions of refinement. A disadvantage to this approach is that it imposes limitations on specifications. According to [82], requiring nondeterministic low level behavior forbids common forms of parallelism for the low level and limits the possible abstractions in the abstract specifications.

### 3.1.3 Secure information flow and data refinement

The notion of data refinement relates an abstract data structure to a concrete data structure [23] as specified by a translation relation. Closing security properties under this kind of refinement is not a viable solution because all possible translation relations must be taken into consideration.

Graham-Cumming and Sanders [43] were to our knowledge the first to consider this problem. They define a refinement notion based on so-called downwards simulations and propose a condition under which a non-interference security property is preserved under this notion of refinement. The security property is defined in terms of a low level equivalence relation, and the condition demands that this relation be preserved for a refinement to be secure. Their notion of refinement is limited in the sense that only refinement of internal data is considered; in other words, they do not consider refinement of input and output data.

Santen et al. [52, 106–108] discussed previously also addresses data refinement. That is, they allow refinements that rename data contained in events of traces. This
3.1 The formal foundation for model-driven information flow security

The notion of data refinement is quite limited because it does not allow one (abstract) event to be refined into a sequence of (concrete) events.

A more general notion of data-refinement of so-called configuration structures is considered by Hutter [57]. An configuration structure has sets of events as its states. The idea is that each state contains the events performed by the system in order to reach that state. A transition relation is therefore (implicitly) defined as the subset relation over states. Hutter defines so-called view refinement in which an event of a configuration structure can be replaced by another configuration structure representing the concrete behavior of that event. This refinement is subject to certain conditions, e.g., that refinements of low level events cannot contain high level events, that together ensure preservation of security.

3.1.4 Secure information flow and composition

If the fact that one or more specifications are secure w.r.t. a security property implies that their composition is secure also, then the security property is preserved under composition, or compositional. The importance of preserving security under composition has long been recognized, and the problem has been studied in a number of papers [17, 83, 85, 87, 130]. We give a brief presentation of these papers.

McCullough is one of the earliest authors to consider the compositionality of security properties. In [85], he observes that the generalized non-interference property [84] is not in general preserved under a notion of parallel composition. However, the property is compositional if there are no mutual feedback loops between systems; that is, if no two systems are allowed to pass messages in both directions. This restriction is of course extremely strong. However, McCullogh shows that the security property called restrictivenss [84] is compositional even in the presence of feedback loops.

In [87], McLean investigates three restricted notions of composition: product, cascade, and feedback. A composition of two systems $S_1$ and $S_2$ is a product if the two systems do not communicate with each other. The composition is a cascade if all outputs of $S_1$ are inputs of $S_2$ and $S_2$ has no other inputs. Feedback composition is essentially the parallel execution of two systems that communicate with each other. McLean shows that properties expressed in his framework are generally preserved under product and cascading, but not under feedback.

Zakinthinos and Lee [130, 131, 133] consider a notion of parallel composition based on simple-hookup [62]. They prove McCullough’s conjecture that non-interference is composable without feedback. In addition they provide a condition that preserves the non-interference property in the presence of feedback.

The work of Mantel [83] is based on the same notion of composition as Zakinthinos and Lee, but his approach is more general. Rather than investigating particular security properties, he presents a general condition (called the generalized un-zipping lemma) which can be used to prove that security properties expressed in the MAKS framework [81] are compositional.

The work of Bossi et.al. [17] is even more general in that they do not “fix” their notion of composition. Instead they define a general operator that captures not only parallel composition, but also other operators of a process algebraic language. They propose a general condition under which security properties are preserved under this notion of composition.
3.2 The method for model-driven information flow security

Model-Driven Security (MDS) is a more recent research field than information flow security. Consequently it is not as extensively studied in the literature. As far as we know, the term MDS was first coined in [10]. Most of the papers addressing MDS consider the specification of access control requirements [10–12, 20, 29, 33, 77, 123]. Perhaps the most notable in this area is the work related to SecureUML [10–12, 77]. SecureUML is an extension of UML for modeling platform independent access control requirements. It is intended that SecureUML be used together with a system design language (e.g., UML class diagrams or UML statecharts). The SecureUML access control requirements can be transformed into platform specific models. In [11], three platforms are considered: Enterprise JavaBeans (EJB), Microsoft Enterprise Services for .NET, and Java Servlets. An advantage of this approach is that these platforms already have access control enforcement mechanisms. However, it is difficult to precisely characterize what it means that a system satisfies the enforcement mechanisms. This means that, although [11] formalizes what it means that a system adheres to an access control model on the platform independent level, they are unable to do so for the platform specific level. No other approach to MDS that we are aware of gives a precise description of what it means that a specification is secure (even for the platform independent models). Consequently, these approaches do not exploit the full potential of MDS.

Fernandez-Medina et. al. [29–32, 119, 123] show how platform independent access control requirements for a conceptual data base model can be expressed in an extension of UML and OCL. However, only an informal description of how to interpret the security constructs of the extension is given. A transformation from the conceptual data base model and its associated access control requirements to an XML database schema is also discussed without being precisely defined in the papers. Similarly, [20] proposes a platform independent model for access control and discuss issues that must be considered in order to transform this into a platform specific model, but no precise characterization of transformation or adherence is given.

Other approaches to MDS that are not specifically related to access-control requirements are [19, 48, 53, 93].

Heldal and Hultin [53] presents an approach in which UML diagrams can be annotated by security (in the sense of confidentiality) requirements. The work also discuss the possibility of transforming annotated UML diagrams into Java code that can be validated w.r.t. confidentiality constraints by the language-based checker Jif (Java information flow).

Nakamura et. al. [93] present a tool framework for web service security. Three levels of abstraction are considered: the operation level, the execution level, and the deployment level. At the operation level, UML models can be annotated with security primitives such as “integrity” or “confidentiality”. Security requirements at the execution level are assumed to be written in so-called deployment descriptors of J2EE. At the deployment level, security requirements of the execution level are bound to specific security infrastructures. Transformation rules between the abstraction levels are discussed, but no precise characterization of adherence or transformations is given.

Haftner et. al. [19, 48] defines the abstract syntax of a domain specific language for the design of inter-organizational workflows. The language supports various cate-
3.3 The method for model-driven policy specification

Policies are rules governing the choices in the behavior of a system [118]. In principle, a security policy may be seen as specification of a system, but in practice, there is an important difference. While a specification is used during the system development process to build a correct system, a policy may also be used after the system has been
developed to configure and ensure that a system is secure. Hence, one of the advantages of policy-based management is that it simplifies and automates the administration of IT environments.

Policies may be classified into obligations, permissions, and prohibitions [59, 128]. Obligations stipulate what a system must do, permissions stipulate what a system may do, while prohibitions stipulate what a system must not do.

Apart from the papers reviewed in the previous section (Sect. 3.2) on the transformation of access control requirements\(^3\), the only related work we are aware of that address policy specification in the setting of MDA is [13, 97, 109].

In [13], Beigi et al. discuss three different types of policy transformations needed in a system managed by the use of policies. In the first scenario, it is assumed that there is a set of static transformation rules for converting policies in terms of high level goals into policies in terms of low level configuration parameters understandable by the system. The idea is to simplify the policy language as seen by the system administrator. In the second scenario, it is assumed that the transformation module holds a table of policies appropriate for the system. The system administrator queries the module with a set of configuration parameters in order to obtain a set of goals that can be achieved given the input. In the third scenario, a case based database or history of system behavior is used to provide an experimental basis for the transformation of high level policies into low level configuration parameters.

In [97], Patz et al. presents the architecture MSME for expressing abstract security policies that can be automatically bound to enforcement mechanisms. Particular attention is given to the problem of resolving policies for a coalition of members that each has its own set of policies. In the MSME architecture, abstract policies are expressed as logical conditions. Patz et al. also discuss how to bind these abstract policies into concrete enforcement mechanisms.

In [109], Satoh et al. consider the transformation of security policies written in the Web Services Security Policy Language (WS-SecurityPolicy) into platform specific configurations of platforms such as IBM WAS, Microsoft WSE, and Apache WSS4J. They observe that a direct mapping from a WS-SecurityPolicy specification to a configuration is complicated to write. They therefore propose a model which can be used as an intermediate step from the transformation of the WS-SecurityPolicy to the configuration.

Other works that are related to MDA in the sense that they use UML for specifying policies are [1, 75, 120].

In [120], the use of UML sequence diagrams for policy specification is considered. The paper argues that sequence diagrams must be extended with customized expressions for deontic modalities to support policy specification.

The reference model for open distributed processing [59] (RM-ODP) establishes concepts for the specification of distributed systems, including a deontic notion of policy. Some papers have attempted to formalize these concepts [1, 75].

Aagedal et al. [1] investigate how the relevant constructs of UML can be used to represent the concepts of RM-ODP and support the specification of enterprise systems. They suggest that the Object Constraint Language (OCL) [125] may be used to specify deontic modalities of policies. E.g., a permission is specified with a pre-condition such that a behavior is allowed if the condition evaluates to true.

\(^3\)Some authors regard access control requirements as policies.
Linington [75] discusses the use of UML for specifying policies in the setting of RM-ODP. He argues that some of the information held by a policy can be expressed by UML, but no firm notational recommendations is given.
State of the art
Chapter 4

Research method

In this chapter, we briefly present a method for technology research and describe how we have applied this method in our research.

4.1 A research method for technology research

The research on which this thesis reports can be labeled technology research. Whereas classical research aims to gain new knowledge about the world, technology research aims to develop or improve existing artifacts, e.g., new algorithms for computers, methods of constructing software, etc. [121].

The development of an artifact is motivated by some kind of need for that artifact. This need gives rise to requirements. The overall hypothesis of technology research is that the new artifact satisfies the requirements.

A common way of verifying that the overall hypothesis is satisfied, is to formulate a set of sub-hypotheses and corresponding predictions whose falsification imply that the overall hypothesis is discredited. Hence, the predictions serve as a basis for gathering research evidence about the validity of the overall hypothesis. According to [86], research evidence is gathered to maximize three things: (A) generalizability, (B) precision, and (C) realism. [86] defines eight research strategies that each have different strengths and weaknesses w.r.t. to (A), (B) and (C).

- Field studies refer to efforts to make direct observations of ongoing systems. “Field studies gain realism at the price of low generalizability and lack of precision” [86]. E.g., this method might be used to study the usability of a groupware program in a work setting by observing how users use the groupware.

- Laboratory experiments are attempts to create and maximize the “essence” of some general class of systems by controlling the extraneous features of the situation. This method is high on precision of measurement. An example of a laboratory experiment is to create a computer network and manipulate the traffic to study how new queuing algorithms in routers affect latency in the network.

- Field experiments are similar to field studies, but with one major difference; the deliberate manipulation of some feature whose effects are to be studied. This method might be carried out in a computer science setting for example by observing the effects of deliberately increased workload of the computer system in a certain workplace.

- Experimental simulations is a laboratory study in which an effort is made to create a system that is like some class of naturally occurring systems. One might for example
study the use of a particular groupware in a created work environment in order to assess attributes of the groupware.

- **Sample surveys** are efforts to get information from a broad and well devised sample of actors. This method is high on generalizability, and might be used in computer science to assess how certain attributes of something (e.g., software, hardware, and programming languages) are considered by a population as a whole.

- **Judgment studies** are efforts to get responses from a selected sample of “judges” about a systematically patterned and precisely calibrated set of stimuli. Judgment studies, as opposed to sample surveys, are considered to be high on precision of measurement, but low on generalizability. This method might be used in a computer science setting, for example to study how a groupware has affected work effectivity by getting precise response from a sample of expert users.

- **Formal theory**. Argumentation based on general formal theory is often high on generalizability. It is not very high on realism of context. In a computer science setting, formal theory might be used for precise argumentation, for example by proving that desirable properties are satisfied by an artifact.

- **Computer simulations** are attempts to model a specific real life system or class of systems. This method might be used in a computer science setting by simulating a real computer network to study congestion in the network.

Regardless of verification strategy, if the overall hypothesis does not hold, then the requirements or the artifact must be revised. Thus, technology research, like classical research, is an iterative process [121].

In summary, technology research can be characterized by the following three phases [121]:

1. **Problem analysis** - The need for new artifacts is identified.
2. **Innovation** - An artifact that satisfies the identified need is constructed.
3. **Evaluation** - Based on the identified need, predictions about the artifact are documented. If the artifact satisfies the need, then this is argued for.

The three phases are illustrated in Fig. 4.1.

Technology research is similar to technology development. According to [121], what distinguishes the two, is that the artifact of the technology research represents new knowledge which is of interest to others. In addition, the knowledge and the artifact must be documented in a way that allows for verification by others.

### 4.2 How we have applied the research method

The method adapted in this thesis is based on the method described in Sect. 4.1. In particular, this means that a highly iterative process was followed, in which the
4.2 How we have applied the research method

artifacts and requirements were constantly improved in light of new insight gained from evaluation. The documentation of the overall problem analysis, innovation, and evaluation phases are found in Chap 2, Chap. 5 and Chap. 7, respectively.

Fig. 4.2 illustrates in more detail the process which was followed in the development of each artifact. For each artifact, we first identified requirements to the artifact. These requirements were based on our goal for developing the new artifacts. Then, we set out to develop an artifact that satisfied the requirements. Finally, we evaluated the invented artifact w.r.t. the identified requirements. As shown in Fig. 4.2, we mainly performed two kinds of evaluation.

1. Checking the artifact against literature, examples, and own understanding. This manner of evaluation is essential for instance when proposing mathematical definitions. E.g., a mathematical definition of security cannot be proven correct even though it is precise and unambiguous. Instead, it has to be checked against examples and own understanding to see if it is useful and intuitive.

2. Examining mathematical properties. Although definitions cannot be proven correct, it is possible to prove that they satisfy desirable properties that are known to be useful. Examples of such properties are compositionality and refinement preservation. Well-structured mathematical proofs are considered a good way of documenting whether properties are satisfied.

The process of evaluation not only led to new insight about the artifact itself, but also about the requirements to the artifact. Hence, as indicated in Fig. 4.2, it was sometimes necessary to revise the requirements to the artifact in light of the insight gained from evaluation.

The process illustrated in Fig. 4.2 was adapted for each of the three artifacts developed in this thesis. In the following, we describe in more detail how each of the three artifacts where evaluated.

### 4.2.1 The formal foundation for model-driven information flow security

For the formal foundation, an important part of evaluation was to check that the foundation was in compliance with established literature on information flow security. For the initial results on the formal foundation, we have (in Sect. 3 of Paper 1 and Sect. 4 of Paper 2), described how these results are in fact generalizations of established results of the literature. Furthermore, in Sect. 3 and the Appendices of Paper 3, we prove that the final version of our foundation is sufficiently expressive to capture many of the information flow properties of the literature.

Countless examples (and counter examples) were constructed to check if the formal foundation satisfied its requirements. During this process, it was frequently the case that the foundation had to be revised because it did not meet its requirements. In certain cases, it was also necessary to revise the requirements to the artifact in light of the insight gained from evaluation. For instance, we initially examined another notion of refinement in addition to the one considered in this thesis. However, this notion of refinement turned out to be too liberal to allow for a useful notion of security preservation. Hence, this notion was discarded from consideration.

The process illustrated in Fig. 4.2 was adapted for each of the three artifacts developed in this thesis. In the following, we describe in more detail how each of the three artifacts where evaluated.

### 4.2.1 The formal foundation for model-driven information flow security

For the formal foundation, an important part of evaluation was to check that the foundation was in compliance with established literature on information flow security. For the initial results on the formal foundation, we have (in Sect. 3 of Paper 1 and Sect. 4 of Paper 2), described how these results are in fact generalizations of established results of the literature. Furthermore, in Sect. 3 and the Appendices of Paper 3, we prove that the final version of our foundation is sufficiently expressive to capture many of the information flow properties of the literature.

Countless examples (and counter examples) were constructed to check if the formal foundation satisfied its requirements. During this process, it was frequently the case that the foundation had to be revised because it did not meet its requirements. In certain cases, it was also necessary to revise the requirements to the artifact in light of the insight gained from evaluation. For instance, we initially examined another notion of refinement in addition to the one considered in this thesis. However, this notion of refinement turned out to be too liberal to allow for a useful notion of security preservation. Hence, this notion was discarded from consideration.
Most of the examples were small, and designed to quickly check that the foundation behaved as intuitively expected. Most of these examples, particularly the counter examples, were later discarded, i.e., not documented in the thesis. Even so, all papers documenting the formal foundation (Papers 1 - 4) contain a number of examples that are intended to show that the foundation is useful and that it satisfies its requirements.

Many of the identified requirements to the formal foundation were in the form of properties that the foundation was intended to exhibit. These properties were formalized in order for us to prove that they were satisfied. All papers on the formal foundation (Papers 1-4) contain a number of proofs that show that the foundation exhibit these properties.

To minimize the possibility of introducing error into the proofs, all proofs were written using Lamport’s proof method [72]. The method is based on hierarchical structures and, as noted by Lamport and experienced by us, it makes it much harder to prove things that are not true.

### 4.2.2 The method for model-driven information flow security

To evaluate the method for model-driven information flow security, we developed an extended example which is intended to demonstrate that the method satisfies its requirements. The example covers all the steps of the method from start to finish, thus the entire method is considered.
4.2 How we have applied the research method

As with the formal foundation, we identified properties that the method was intended to exhibit. These properties were formalized and verified by proofs written using Lamport’s proof method. Some of these properties were satisfied by virtue of the method being supported by the formal foundation for model-driven information flow security. However, some properties that were not captured by the foundation were also formalized and proved. This is documented in the Appendices of Paper 5.

4.2.3 The method for model-driven policy specification

Part of the work on the method for model-driven policy specification was conducted in the EU-project S$^3$MS [112]. Many of the examples that we used to verify our method were provided by the industrial partners of the project. We believe that more realism was gained in the evaluation by using examples not created by ourselves. Some of the examples which were provided by the industry partners are documented in Sect. 2 - 4 of Paper 6.

As with the two other artifacts, some essential properties of the method were proven to hold. This not only means that we can claim with reasonable confidence that the method does indeed satisfy these properties, but it also means that many parts of the method had to be formalized in sufficient detail to allow for verification by proof.

To demonstrate the feasibility of our method, a tool which automates some of the method steps was created. This tool is documented in Sect. 5.4.1.
Chapter 5

Achievements: the overall picture

In this chapter, we give an overview of the artifacts resulting from our work and how they fit together.

5.1 Overall picture

The main contributions of this thesis are:

(A) The formal foundation for model-driven information flow security;

(B) The method for model-driven information flow security;

(C) The method for model-driven policy specification.

Artifact (B) and artifact (C) are both methods to MDS. One of the main differences between the two methods is the kind of security requirements they address. That is, artifact (B) addresses secure information flow properties while artifact (C) addresses prohibition policies. Because working with secure information flow properties is theoretically more challenging than working with prohibition policies, artifact (B) is supported by a formal foundation (i.e., artifact (A)). Note that artifact (C) is also supported by a formal foundation, but we do not call it an artifact in its own right because it is not as extensive as the foundation for secure information flow properties.

Secure information flow properties are often generic in the sense that a single property can be applied to many different systems. Prohibition policies, however, although less expressive than information flow security properties, are often intended to be written for particular software systems. Thus, secure information flow properties and prohibition policies have different roles, and may be seen as complimentary.

Another reason why artifact (B) and artifact (C) may be seen as complimentary, is that artifact (B) addresses transformation of the system specification, while artifact (C) addresses transformation of the security requirement (the prohibition policy). This is illustrated in Fig. 5.1. In the scenario on the left hand side, the system specification and the security requirement (i.e., the prohibition policy) is not developed together. The system specification may for instance be legacy code, or a system developed by someone other than the policy writer. In this case, the abstract specification may not be available, and adherence can only be shown between the concrete system specification and the concrete policy. In the scenario illustrated on the right hand side of Fig. 5.1, the system specification and the security requirement are intended to be developed
Achievements: the overall picture

Abstract prohibition policy

Concrete prohibition policy

System specification

Abstract system specification

Concrete system specification

Secure information flow property

transformation/refinement

adherence

Figure 5.1: Transformation of security requirement (LHS) and transformation of system specification (RHS)

Figure 5.2: Combination of the two methods

together. However, due to the generic nature of secure information flow properties, the same property can be used at the abstract and the concrete level. Thus only transformation of the system specification is considered. Note that in this scenario, one may introduce new security requirements to the concrete level that are not considered at the abstract level, but we do not regard these additional requirements as transformations of the abstract security requirements. Therefore, this is not illustrated in Fig. 5.1.

Because the methods of artifact (B) and artifact (C) are complimentary, they may be used in combination. In Fig. 5.2, we have illustrated how this may be achieved. Here, it is intended that the system specification and the security requirements (in the form of both prohibition policies and secure information flow properties) are developed together. Artifact (C) may be extended to take this into account by defining adherence between the abstract system specification and the abstract prohibition policy. The transformation from the abstract level to the concrete level should preserve adherence to both the policy and the secure information flow property. Thus adherence need not be reestablished at the concrete level and the full benefit of MDS is exploited.

5.2 The formal foundation for model-driven information flow security

The purpose of the formal foundation is to precisely characterize the key aspects of modular and stepwise development of secure software systems. The formal foundation enables rigorous and precise reasoning about the security of specifications and preservation of security under transformation, composition, and refinement.
5.2 The formal foundation for model-driven information flow security

The formal foundation is documented in the Papers 1 - 4. It consists of the following main (sub)artifacts:

- a security framework for specifying secure information flow properties that are preserved under refinement;

- conditions under which security properties defined in the framework are preserved under transformation and composition.

In the following, we give a brief overview of these (sub)artifacts.

5.2.1 The security framework

Secure information flow properties are requirements on the allowed flow of information between security domains. For simplicity, but without lack of generality, it is conventional to consider two security domains only – one low level domain and one high level domain – together with the requirement that information must not flow from the high level to the low level domain.

A secure information flow property defines what it means that information flows from one domain to another. The underlying idea is to require that the set of possible behaviors of a system that could have resulted in a given low level observation, must include a behavior that prevents the low level observer from asserting that confidential behavior of the high level domain has or has not occurred.

The security framework defines a general schema for specifying secure information flow properties. The schema is parameterized by the behavior that must be present in a system to prevent the low level observer from asserting that confidential behavior has or has not occurred.

The first version of the framework (presented in Sect. 4 of Paper 2), is a generalization of a framework proposed by Mantel in [81]. Unlike the framework of Mantel, our framework is defined in a more expressive semantic model based on STAIRS [50,51,103]. STAIRS is a formal approach to system development with UML 2.1. sequence diagrams that supports an incremental and modular development process. Although STAIRS is developed for UML sequence diagrams, the semantic model of STAIRS is independent of the sequence diagram notation. The STAIRS semantics can therefore be used as a model for other languages as well. We make use of the expressibility offered by the STAIRS model to show that all properties defined in our framework are preserved under the STAIRS notion of refinement.

The second version of the framework (presented in Sect. 3 of Paper 3 and Sect. 4 of Paper 4) is a simplification of the first version of the framework. The simplification makes the framework easier to work with. We show that, although the framework is simpler than the first framework, it is nevertheless sufficiently expressive to capture many of the standard secure information flow properties presented in the literature.

5.2.2 Preservation of security under transformation and composition

Semantically, a \textit{transformation} may be seen as a mapping from specifications described in an abstract vocabulary to specifications described in a concrete vocabulary of a finer granularity. A formal characterization of the semantics of transformations is given in
Sect. 4 of Paper 3 (preliminary definitions are given in Sect. 5 of Paper 1 and Sect. 4 of Paper 2).

Security properties are not in general preserved under arbitrary transformations. The reason for this is that the concrete specifications are usually more detailed than their abstract equivalents. Hence, the low level user may make more observations and assert more about the behavior of the high level domain on the concrete level than on the abstract level. In Sect. 4 of Paper 3, we present a general condition under which security properties defined in our framework are preserved under transformation. The condition essentially states that the transformation must preserve low level equality and confidential behavior. I.e., if two behaviors result in the same low level observation, then these behaviors must be translated to behaviors which result in the same low level observations also. In addition, if a behavior is not considered confidential, it cannot be transformed into behaviors which are considered confidential. The general condition may be instantiated in order to obtain specific conditions that can be used to check that a given transformation preserves a given security property.

In Sect. 5 of Paper 3, we define a general notion of composition, and show that this notion is sufficiently expressive to capture standard composition operators such as parallel composition, sequential composition, and nondeterministic choice. In Sect. 5 of Paper 3, we also propose a general condition under which secure information flow properties defined in our framework are preserved under composition. This condition is in fact a generalization of the condition under which security is preserved under translation.

Our foundation offers the advantage that composition may be used together with transformation and refinement in such a way that security is preserved. This is because all security properties defined in our framework are preserved under refinement.

### 5.3 The method for model-driven information flow security

The method for model-driven information flow security is described in Paper 5. The method defines a stepwise and modular development process in which UML inspired state machines are used to specify secure systems. The method has seven main steps which are supported by the formal foundation described in Sect. 5.2.

In step I, the user of our method (i.e., a software developer) specifies the system architecture using UML composite structures. The system architecture is an overview of the components of the system and their associated communication channels. The user then partitions the system into security domains by labeling the system architecture with security relevant annotations.

In step II, the user selects a set of secure information flow properties (typically from a library) that the system must adhere to. The secure information flow properties selected are assumed to be defined in the security framework (described in Sect. 5.2.1) which ensures that the adherence to the properties is preserved when underspecification is reduced by refinement in later steps.

In step III, the user specifies each component of the system architecture using our UML inspired state machines. The state machine notation provides constructs for specifying both design choices and choices that must be offered by the components of the system. The specified state machines may reference events that are already
5.3 The method for model-driven information flow security

Figure 5.3: Use cases of the MDSTool

provided in a predefined event library. In this step, new event specifications may also be uploaded to the event library.

In step IV, the user verifies that the state machine based specification adheres to the security properties selected in step II. There are many techniques and methods that can be used for this purpose. We do not go into details on these. However, we provide a precise characterization of what it means for a system specification to adhere to a secure information flow property. This characterization enables rigorous adherence verification.

In step V, the user refines the specification by removing alternative design decisions until all design decisions are decided. The validity of adherence is guaranteed to be preserved under this kind of refinement.

In step VI, the (abstract) state machine specification of step V is transformed into a more concrete specification by substituting the event references of the abstract specification by their definitions in the event library. We call this an event transformation.

In step VII, the user verifies that the event transformation of step VI preserves adherence to the security property selected in step II. We present conditions that can be used to check that the transformation preserves adherence.

The method is described in detail in Paper 5. In the paper, we provide a formal semantics for UML inspired state machines, and we give a formal characterization of the syntax and semantics of event transformations. We also show that the event transformations satisfy some desirable properties, and we define a condition (based on the formal foundation described in Sect. 5.2) under which these transformations preserve secure information flow properties.
5.3.1 Tool support

It is intended that the method should be supported by a tool (which we call the MDSTool). Parts of the tool has already been developed. In this section, we discuss the functionality of the tool, and we explain what has already been done and what remains to be done.

Fig. 5.3 shows an UML use case diagram that specifies the main functionality of the MDSTool. These use cases are:

(UC1) Select transformation Select a predefined transformation from a transformation library.

(UC2) Create transformation Create a new transformation.

(UC3) Select source specification Select the specification which represents the source / input to the transformation.

(UC4) Execute transformation Execute the selected transformation, i.e., apply the selected transformation to the selected source specification.

(UC5) Select security property Select a predefined security property that the source specification should be in adherence to.

(UC6) Verify adherence Verify that the selected source specification adheres to the selected security property.

(UC7) Verify transformation Verify that the selected transformation is adherence preserving w.r.t. to the selected security property.

The use cases (UC1) - (UC4) have been implemented in the tool that supports our third artifact, the method for model-driven policy specification. This functionality can therefore be used by the MDSTool as well. However, the transformation from abstract to concrete state machines (which is intended to be predefined for the user) must be written. In Sect. 5.4.1 we discuss (UC1) - (UC4) in more detail.

The tool for the third artifact has been written using the programming languages Java and Prolog. Java was used to create the graphical user interface of the tool, and Prolog was used to define the transformations which can be selected by the user. In Sect. 5.4.1 we discuss (UC1) - (UC4) in more detail.

The use cases (UC5) - (UC7) have not been implemented yet because further development on the formal foundation is needed to enable this functionality. W.r.t. use case (UC6), there are some few existing tools for enforcing adherence to secure information flow properties [35, 92]. The tool of Foccardi et al. [35] is based on bisimulation for a process algebra, whereas the tool of Myers [92] is based on type checking. The latter approach makes use of inference rules to inductively characterize those specifications that are secure. This is approach is quite elegant, and the kind of inference rules used in type checking can quite directly be implemented in Prolog (which is the language we have used for writing transformations). Furthermore, the same approach can be used as a basis for implementing use case (UC7). I.e., instead of type checking specifications, this use case would type check transformations to ensure adherence preservation. For these reasons, the uses cases (UC5) - (UC7) should be based on a type checking approach similar to [92].
5.4 The method for model-driven policy specification

The method for model-driven policy specification is presented in Paper 6. The method defines a process for using UML sequence diagrams to specify high level security policies that can be enforced by execution monitoring mechanisms (EM mechanisms). EM mechanisms work by monitoring execution steps of some system, and terminating the system’s execution if it is about to violate the security policy being enforced.

The security policy which is enforced by an EM mechanism is often specified by a state machine that describes exactly those sequences of security relevant actions that the system being monitored is allowed to execute. Such EM mechanisms receive an input whenever the system is about to execute a security relevant action. If the state machine of the EM mechanism has an enabled transition on a given input, then the current state is updated according to where the transition lands. If the state machine has no enabled transition for a given input, then the system is about to violate the policy being enforced. It is therefore terminated by the EM mechanism.

UML sequence diagrams, unlike state machines, are partial in the sense that they typically don’t tell the complete story. There are normally other legal and possible behaviors that are not considered within the described interaction. In particular, sequence diagrams explicitly describe two kinds of behavior: behavior which is positive in the sense that it is legal, valid, or desirable, and behavior which is negative meaning that it is illegal, invalid, or undesirable. The behavior which is not explicitly described by the diagram is called inconclusive meaning that it is considered irrelevant for the interaction in question.

The construct of specifying explicit negative behavior which is offered by UML sequence diagrams makes them suitable for specifying the kind of policies that can be enforced by EM mechanisms. The reason for this is that the only kind of policies that can be enforced by EM mechanisms are so-called prohibition policies, i.e., policies that specify what a system is not allowed to do.

Because there are no EM mechanisms that we are aware of that can enforce sequence diagrams, of method is supported by a tool which transforms sequence diagrams
Achievements: the overall picture

Figure 5.5: Illustration of the output of the three method steps

into state machines that can be enforced by EM mechanisms. The transformation from sequence diagrams to state machines may be parameterized by mapping rules for translating abstract to concrete behavior. We show how such mapping rules can be specified using UML sequence diagrams patterns, i.e., sequence diagrams that may contain so-called meta variables.

In summary, our method has three main steps:

**Step I** The user of our method receives a set of policy rules written in natural language, and formalizes these using UML sequence diagrams.

**Step II** The user writes/selects a set of transformation rules (expressed in UML sequence diagrams) from a transformation library, and applies these using a tool to automatically obtain a low level intermediate policy (also expressed in UML sequence diagrams).

**Step III** The tool automatically transforms the low level intermediate policy expressed in UML sequence diagrams into a state machine that governs the behavior of an EM mechanism.

In Fig. 5.5, the output of each of the three steps is illustrated.

In Paper 5, we provide a formal characterization of the transformation from UML sequence diagrams to state machines. We also formally define what it means that a system adheres to a UML sequence diagram policy and to a state machine policy.

5.4.1 Tool support

To support our method, we have developed a proof-of-concept tool (called the TTool for transformation tool). The tool enables its user to select and execute transformations. For generality, the tool may offer other transformations than the transformation from sequence diagrams to state machines. All transformations that can be selected by the user are written in the language Prolog. Prolog is a declarative programming language based on logical inference rules. The reason we have chosen Prolog is that it is relatively straightforward to translate the mathematical characterization of the transformations as described in this thesis into Prolog. This gives added confidence that the implementation of the transformations is correct.

The Prolog transformation from UML sequence diagrams to state machines takes a simple textual representation (called the STAIRS syntax) of UML sequence diagrams as input. The STAIRS syntax is the representation over which both the operational and the denotational semantics of sequence diagrams is defined according to STAIRS.
5.4 The method for model-driven policy specification

It is intended that the TTool should be used in combination with a sequence diagram editor. To test the TTool, we used the SeDi sequence diagram editor [74] as a plug-in for the case tool IBM Rational Software Modeler (RSM) [58]. Fig. 5.6 shows an example of a sequence diagram that is specified in the SeDi editor.

UML models of the SeDi editor are stored in the XMI (XML Metadata Interchange) format. Because the predefined transformation of the TTool takes sequence diagrams represented in the STAIRS syntax as input, it was necessary to convert the XMI models into STAIRS syntax. To create this conversion, we used a language called MOFscript [117]. The reason we chose this language is that it is specifically developed for writing transformations from models (XMI files) to text.

In Fig. 5.7, we have illustrated how SeDi/RSM, the MOFscript engine, and the TTool are related. Here, the XMI representation of the sequence diagram is given as input to the MOFscript engine which converts the XMI into STAIRS syntax. The STAIRS syntax is given as input to the TTool. Note that the final output is a state machine which is either represented in the XMI format or in the so-called ConSpec language [3]. A ConSpec specification is essentially a guarded command representation of a state machine. The reason why this language is supported, is that policy enforcement monitors have been created for state machines that are represented in this language [24].

Figure 5.6: Example of a sequence diagram specified in the SeDi editor
Achievements: the overall picture

Figure 5.7: Illustration of the TTool setup

Figure 5.8: Graphical user interface of the TTool

Fig. 5.8 illustrates the graphical user interface of the TTool. It has three fields of input:

Source specification This text field contains the path to the file that the transformation takes as input.

Target specification This text field contains the path the output file of the transformation.

Transformation This field is a list of paths to Prolog files that specify the transformation.

In addition, the tool has two buttons:

Transform When this button is pressed, the TTool will read the source specification, execute the specified transformations, and produce the target specification.

Make transformation When this button is pressed, a new window will appear (see Fig. 5.10) which enables the user to customize transformations.

The make transformation button, enables the user to create new transformations based on mapping rules. A mapping rule is a pair of two sequence diagram patterns (i.e., sequence diagrams that may contain meta variables: one left hand side pattern and one right hand side pattern. When a mapping rule is applied to a sequence diagram, all parts of the diagram that match the left hand side pattern are replaced by the right hand side pattern.
The mapping rules enable the user to customize the transformation from sequence diagrams to state machine by, e.g., to change the granularity of messages in the sequence diagram.

Fig. 5.9 illustrates the process of making a new transformation. First, a set of sequence diagram patterns are specified in the SeDi editor. Then, the MOFscript engine is used to convert the XMI representation of the sequence diagram patterns into STAIRS syntax. Finally, the sequence diagram patterns (represented in the STAIRS syntax) are given as input to the TTool which then creates a new (Prolog) transformation based on the sequence diagram patterns.

The TTool window for specifying new transformations is shown in Fig. 5.10. The window has two sources of input:

**Rule list** This is a list of left hand side and right hand side sequence diagram patterns.

**Transformation file** This is the path to the file which contains the transformation which is produced from the rule list.

In addition, the window has four buttons:

**Add LHS** Adds a new left hand side diagram pattern to the rule list.

**Add RHS** Adds a new right hand side diagram pattern to the rule list.

**Set file** Sets the file path to the transformation which is produced from the rule list.

**Make transformation** Reads all transformation rules in the rule list and creates a new (Prolog) transformation which is written to the specified transformation file.
Figure 5.10: Graphical user interface of the create new transformation window
Chapter 6
Overview of research papers

The main results of our work are documented in the papers presented in Part II. In this chapter, we provide publication details of each of these papers.

6.1 Paper 1: Information Flow Property Preserving Transformation of UML Interaction Diagrams

Authors: Fredrik Seehusen and Ketil Stølen.

Publication status: Technical report STF40 A03066, SINTEF Telecom and Information. The research report presented in this thesis is a revised and expanded version of the article published in the proceedings of the 11th ACM Symposium of Access Control Models and Technologies (SACMAT’06) [114].

My contribution: I was the main author, responsible for about 90% of the work.

Main topics: This paper constitutes the initial development of the formal foundation for model-driven information flow security. In the paper, we define an information flow security property in the STAIRS semantics, and show that it is preserved under the STAIRS notion of refinement. We also define a (semantic) notion of transformation and define a condition under which the property is preserved under this notion.

6.2 Paper 2: Maintaining Information Flow Security under Refinement and Transformation

Authors: Fredrik Seehusen and Ketil Stølen.

Publication status: Technical report A311, SINTEF Information and Communication Technologies. The research report presented in this thesis is a revised and expanded version of the article published in the proceedings of the 4th International Workshop on Formal Aspects in Security and Trust (FAST’06) [115].
My contribution: I was the main author, responsible for about 90% of the work.

Main topics: In this paper, we generalize the results of Paper 1 by defining a security framework for specifying security properties that are preserved under the STAIRS notion of refinement. The framework is a generalization of a framework proposed in [81]. The paper also proposes a general condition under which security properties defined in the schema are preserved under a (semantic) notion of transformation.


Authors: Fredrik Seehusen and Ketil Stølen.

Publication status: Accepted for publication in Journal of IET Information Security.

My contribution: I was the main author, responsible for about 90% of the work.

Main topics: In this paper, we simplify the security framework presented in Paper 2. We show that the (simplified) framework captures many of the standard security properties of the literature, and that all security properties defined in the framework are preserved under the STAIRS notion of refinement. We also define a general notion of translation and a general notion of composition, and propose general conditions under which security properties defined in our framework are preserved under these notions.


Authors: Fredrik Seehusen, Bjørnar Solhaug and Ketil Stølen.

Publication status: Published in the Journal of Software and Systems Modeling (SoSyM) [113].

My contribution: I was one of the main authors, responsible for about 45% of the work.

Main topics: In this paper, the security framework of paper 3 is explained in more detail. The paper also provides a detailed discussion on the expressibility of the semantic model of STAIRS, and on how this expressibility may be exploited to define secure information flow properties and policies that are preserved under refinement.
6.5 Paper 5: A Method for Model-Driven Information Flow Security

Authors: Fredrik Seehusen and Ketil Stølen.


My contribution: I was the main author, responsible for about 90% of the work.

Main topics: In this paper, we present a model-driven method for developing software systems that are secure w.r.t. information flow properties. The method enables its users to specify software systems using UML inspired state machines, to select secure information flow properties that the specification must adhere to, and to refine and transform the specification in such a way that adherence to the selected security properties is preserved. The paper provides a formal semantics for UML inspired state machines. The paper also provides a formal characterization of a (syntactic) notion of transformation and shows how these transformations can be semantically interpreted. Based on the formal foundation, a condition under which these transformations preserve secure information flow properties is proposed.

6.6 Paper 6: A Transformational Approach to Facilitate Monitoring of High Level Policies

Authors: Fredrik Seehusen, Mass Soldal Lund, and Ketil Stølen.

Publication status: Technical report A11356, SINTEF Information and Communication Technologies. The research report presented in this thesis is a revised and expanded version of the short paper published in the proceedings of the 2008 IEEE Workshop on Policies for Distributed Systems and Networks (POLICY’08) [116].

My contribution: I was the main author, responsible for about 90% of the work.

Main topics: In this paper, we present a model-driven method for specifying high level policies that can be enforced by execution monitoring mechanisms. The method has three main steps: (1) the user of our method formalizes a set of policy rules using UML sequence diagrams; (2) the user selects a set of transformation rules from a transformation library, and applies these using a tool to automatically obtain a low level policy (also expressed in UML sequence diagrams); (3) the tool automatically transforms the low level policy expressed in UML sequence diagrams into a UML state machine that governs the behavior of a runtime policy enforcement mechanism. We believe that the method is both easy to use and useful since it automates much of the policy formalization process.
Chapter 7

Discussion

In Chap. 2, we defined success criteria that should be fulfilled by each invented artifact to reach our goal successfully. In the following, we discuss to what extent each success criterion is satisfied by the invented artifacts. We also discuss important design decisions and the literature which is most closely related to this thesis.

7.1 The formal foundation for model driven information flow security

Success criterion 1 The foundation should precisely define what is meant by a specification.

In all the attached papers, a precise notion of specification based on the STAIRS [50] trace model is defined. Trace semantics is a well known model that has proven to be useful. This criterion is therefore fulfilled.

Success criterion 2 The foundation should precisely define what is meant by information flow security.

The security framework developed in Sect. 3 of Paper 3 and Sect. 4 of Paper 4 (preliminary versions are given in Sect. 3 of Paper 1 and Sect. 4 of Paper 2) enables the precise specification of many secure information flow properties, including well known secure information flow properties of the literature. This criterion is therefore fulfilled.

Success criterion 3 The foundation should precisely define what it means that security is preserved under refinement.

In Sect. 3 of Paper 3, we prove that all secure information flow properties defined in our framework are preserved under the STAIRS notion of refinement. This criterion is therefore satisfied.

Success criterion 4 The foundation should precisely define what it means that security is preserved under transformation.
In Sect. 4 of Paper 3, we develop a semantic characterization of transformations (preliminary definitions are given in Sect. 5 of Paper 1 and Sect. 4 of Paper 2). We also define a condition under which security properties are preserved under transformation. The condition is general in the sense that it applies to arbitrary transformations and arbitrary security properties defined in our security framework. By instantiating the general condition, one obtains a specific condition that can be used to check/prove that a given transformation preserves a given security property.

**Success criterion 5** The foundation should precisely define what it means that security is preserved under composition.

In Sect. 5 of Paper 3, we define a general notion of composition. We show that this notion is sufficiently expressive to capture many standard composition operators, e.g., parallel composition, sequential composition, and nondeterministic choice. We also define a general condition under which security properties defined in our framework are preserved under composition. This condition may be seen a generalization of the condition under which security properties are preserved under transformation. The success criterion is therefore fulfilled.

### 7.2 The method for model driven information flow security

**Success criterion 6** The method should enable security analysis at a high level of abstraction.

Security analysis at a high level of abstraction is discussed and explicitly defined as a step in our method (Sect. 6 of Paper 5). Therefore this criterion is satisfied.

**Success criterion 7** The method should support preservation of security under refinement and transformation.

Preservation of security under refinement and transformation is discussed and explicitly specified as steps in our method (Sect. 7-Sect. 9 of Paper 5). Our method therefore fulfills this success criterion.

**Success criterion 8** Be easy to understand and employ by the users of the method (which we assume are software developers).

We believe that this criterion is not completely fulfilled because the method is not yet supported by a tool. However, once such a tool is available, we believe that the method should be easy to use and employ because it is very similar to a traditional software development method. The main difference is that the user has to partition the system into security domains and select a set of security properties that the system must adhere to. No understanding of the underlying formal foundation should be required from the users point of view.

**Success criterion 9** The method should be supported by tool.

Our method is not currently supported by a tool, but the functionality of such a tool is discussed in Sect. 5.3.1.
7.3 The method for model driven policy specification

Success criterion 10  The method should support the specification of policies at a high level of abstraction.

Since the method allows the user to select or create mapping rules (see Sect. 3 of Paper 6) for translating abstract to concrete behavior, the method enables the specification of policies at a high level of abstraction. Therefore this criterion is satisfied.

Success criterion 11  The method should support transformation of high level policies to low level policies.

The mapping rules (see Sect. 3 of Paper 6) for translating abstract to concrete behavior induce a transformation from high to low level policies that can be automated. This success criterion is therefore fulfilled.

Success criterion 12  The method should facilitate automatic enforcement of low level policies.

In Sect. 4 of Paper 6, we define a transformation from policies expressed in UML sequence diagrams to policies expressed in state machines. These state machines can be used to govern the behavior of an execution monitoring mechanism that enforces policies automatically. This success criterion is therefore fulfilled.

Success criterion 13  The method should be easy to understand and employ by the users of the method (which we assume are software developers).

We believe that the method is easy to understand and employ because specifying policies should not be any harder than specifying software systems. In fact, it may be easier to specify policies because policies are usually smaller and simpler than specifications of software systems. In addition, since policies are expressed in UML sequence diagrams, the policy language should be known to a large number of software developers. Therefore this criterion is fulfilled.

Success criterion 14  The method should be supported by tool.

A proof of concept tool has been developed. This tool is documented in Sect. 5.4.1. Therefore this criterion is fulfilled.

7.4 Design decisions

A number of design decisions have been made during the development of the artifacts of this thesis. In this section, we discuss the most important design decisions.
7.4.1 The formal foundation for model-driven information flow security

Why information flow security?

Security is a “horizontal” aspect of software development that affects nearly every part of a system [77]. Many security incidents are related to issues that are addressed by well established areas of computer science such as usability and program correctness. Should security experts assume that the issues which are addressed in other well established areas of computer science are solved, or should security experts address these issues as well? The answer is probably that security experts need to do a little bit of both.

In order to focus on issues that are unique for the area of security, we have not addressed properties such as usability and safety. Instead, we have restricted attention to secure information flow properties. The main reason for considering secure information flow properties, is that the problem of preserving these properties under refinement and transformation has not been adequately addressed in the literature due to lack of expressibility in most formalisms.

Although information flow properties can be used to express integrity properties [82], the main focus of this thesis has been on confidentiality. The remaining aspect of security, availability, has been outside the scope of this thesis.

Why use STAIRS?

UML is the de-facto modeling language of the MDA framework and it is widely known by the software industry. It was therefore important for us to take this language into consideration. Many attempts have been made to formalize various parts of UML. We have chosen a formalization which is based on the STAIRS approach [50, 51, 103]. The main reasons for this choice is that (1) STAIRS is faithful to the UML 2.x standard, (2) STAIRS supports a compositional and incremental software development process, and (3) the STAIRS semantics is sufficiently expressive to distinguish between two kinds of nondeterminism which are needed to preserve security under refinement.

There are no other approaches that we are aware of that satisfy all three of these conditions.

7.4.2 The method for model-driven information flow security

Why UML inspired state machines?

Although STAIRS provides a formal semantics for UML sequence diagrams, we have adapted the STAIRS semantics for UML inspired state machines as well. The reason for this is that we wanted also to consider a language that was closer to the implementation level than UML sequence diagrams. Of the two languages, UML sequence diagrams is a natural starting point. The reason for this is that sequence diagrams characterize example scenarios, or snapshots of behavior in a period of time. This is not the case for state machines which describe complete behavior. State machines are therefore well suited as a second step in the software development process.

UML includes diagrams for describing both the structure and the behavior of sys-
7.4 Design decisions

tems. We mainly consider behavioral diagrams since security requirements are usually requirements on the behavior of systems. Therefore it is more interesting to analyze diagrams that describe behavior than diagrams that describe structure.

For behavior, UML includes, in addition to state machine diagrams and sequence diagrams, so-called activity diagrams and communication diagrams. To limit the scope of the thesis, we do not consider these as sequence diagrams are more general than communication diagrams, and state machines are similar to activity diagrams.

Why not use a standard model transformation language?

In our method, we consider transformations that replace events in state machines by other state machines that describe the behavior of these events at a lower level of abstraction. These transformations are simply specified by state machines whose name equals the events that should be replaced under transformation.

An alternative would be to consider model transformations between UML models that can be specified by model transformation languages such as QVT [45] that are often associated with the MDA framework. This alternative was dismissed because it would require that we give a formal semantics of UML models in order to reason about security preservation. The UML meta model is too big and complicated to make this a reasonable alternative. Since we have followed the STAIRS approach, where the semantics of UML diagrams is defined over an abstract syntax which is much simpler than the UML meta model, model transformation languages such as QVT were not considered a good option.

7.4.3 The method for model-driven policy specification

Why prohibition policies?

The method for model-driven policy specification addresses so-called prohibition policies, i.e., rules that stipulate what an application is not allowed to do. The main reason for this is that we wanted to consider a notion of security which is more easily enforced by a security mechanism than information flow security. Indeed, prohibition policies correspond to so-called safety properties which are exactly the kind properties than can be enforced by runtime monitoring mechanisms. Neither information flow properties, or the two other types of policies, obligation and permission policies, can be enforced by runtime monitoring mechanisms.

Why UML sequence diagrams?

In our method, we use UML sequence diagrams in order to specify prohibition policies. The main reason for this is that UML sequence diagrams (unlike, e.g., state machines) are partial specifications that describe snapshots of behavior in a period of time. This is useful because policies are partial statements that do not typically characterize all aspects of the behavior of an application. Furthermore, a consequence of this is that UML sequence diagrams have constructs for both explicitly specifying positive behavior which should be allowed and negative behavior which should not be allowed. All behavior which is not explicitly described is interpreted as inconclusive. This is in contrast to complete specifications which only describe positive behavior (all other
behavior is assumed to be negative). The construct for specifying explicit negative behavior is useful for specifying prohibition policies, because these policies stipulate what an application is not allowed to do. Indeed, policies expressed in terms of explicit negative behavior are often simpler and more readable than policies expressed as complete specifications.

7.5 Related work

In this section, we discuss the papers that are most closely related to our work. For a more general discussion of related work, the reader is referred to Chapter 3 or the related works sections in the attached papers.

7.5.1 Thomas Santen, Maritta Heisel and Andreas Pfitzmann: Confidentiality Preserving Refinement

In their initial paper on confidentiality preserving refinement [52], Santen et. al. considers CSP (Communication Sequential Processes [55]) extended with a probabilistic choice operator. In CSP, refinement is defined as the set inclusion on the chosen semantics of processes (e.g., traces or failures). That is, a process \( A \) is a refinement of a process \( B \), written \( A \sqsubseteq B \), iff

\[
\llbracket A \rrbracket \subseteq \llbracket B \rrbracket
\]

where \( \llbracket P \rrbracket \) yields the semantics of process \( P \).

In order to express difference in the granularity of abstract and concrete data, Santen et. al. generalize the CSP notion of refinement into what they call behavioral refinement. Behavioral refinement is defined in terms of a so-called retrieve relation \( R \) that maps concrete data to abstract data. According to their definition, a process \( A \) is a refinement of process \( B \) w.r.t. the retrieve relation \( R \), written \( A \sqsubseteq_R B \), iff

\[
R(\llbracket A \rrbracket) \subseteq \llbracket B \rrbracket
\]

where \( R(D) \) maps the set \( D \) to the union of the sets of images of members of \( D \) under \( R \).

The basis of their approach is to augment a specification of the intended behavior of a system – given in terms of a CSP process – by a window that models the possible observations an adversary may make about the system. They assume that the adversary know the structure of the system. Therefore, the adversary may use that knowledge to derive information about the internal states of the system from the observations of the system.

They define confidentiality in terms of a so-called indistinguishability relation on traces. I.e., for a system to be secure, the system must be designed in such a way that the data the window provides to an observer cannot be used to distinguish data it internally stores and that it should keep confidential.

Santen et.al. defines a notion of confidentiality-preserving refinement that must satisfy two conditions: (1) each confidentiality-preserving refinement must be a correct behavioral refinement, and (2) on the concrete level, it must not be possible to distinguish more traces than on the abstract level via the respective windows.

Santen et.al. continue their work in [108], where they investigate the condition under which their notion of confidentiality-preserving refinement is compositional, i.e., the
7.5 Related work

condition under which refining a subsystem of a larger system yields a confidentiality-preserving refinement of the larger system. They argue that their notion of confidentiality-preserving refinement is not, in general, compositional, but state a condition under which it is.

The notion of composition considered is CSP parallel composition. A counterexample that shows that confidentiality-preserving refinement is not compositional is given. The reason why it is not compositional is because, according to their definition, more observations are possible from a composed system, than from its parts. Based on this insight, they propose a condition under which confidentiality preserving refinement is compositional.

As far as we are aware of, [52, 108] constitutes the work which is closest related to our formal foundation for model-driven information flow security. The reason for this is that the notions of information flow security, refinement, composition, and a notion similar to our notion of transformation (the retrieve mapping) are all considered in a unified framework. Despite this similarity, there are many differences between their work and ours. First, they work in probabilistic setting whereas our work is in a possibilistic setting. This means that their semantic foundation differs from ours. Second, they consider a much more restricted notion of security than we do. In fact, they do not precisely define what it means that a system is secure. Instead, they define an important aspect on which many security properties depend, namely the indistinguishability relation, but not how this is used in order to define security. Third, although their notion of behavioral refinement is similar to our notion of refinement modulo transformation, our notion of translation is more general than the data refinement considered by Santen et. al. because we map (abstract) traces to (concrete) trace sets as opposed to (abstract) events to (concrete) event sets. Without this generality, we would not have been able to characterize some of the examples of the papers in Part II, (e.g., the example in Sect. 4.1 of Paper 3). Forth, Santen et. al. consider only one notion of composition (CSP parallel composition) whereas we take a more general approach. That is, rather than restricting attention to a particular composition operator, we define a general notion of composition which can be instantiated to obtain specific composition operators (including parallel composition).

Santen continues his research on confidentiality preserving refinement in [106, 107] by defining a general form of probabilistic security properties and proposing conditions under which this notion is preserved under his notion of behavioral refinement. However, in this work, he takes an approach which is unlike any approach that we are aware of. Instead of demanding that all refinements of a secure specification must be secure, he demands that there must exist a secure refinement of a specification. The benefit of this approach is that it avoids the so-called refinement paradox. However, a drawback is that care must be taken to construct a secure implementation of the specification.

7.5.2 David Basin, Jürgen Doser, and Torsten Lodderstedt: SecureUML

In a series of papers ( [10–12, 77]), Basin et.al. develop an approach to Model-Driven Security (MDS). In particular, they propose a schema for integrating security requirements into system modeling languages. The schema is parameterized by three languages:
1. a **security modeling language** for expressing security requirements;

2. a **system design modeling language** for constructing design models; and

3. a **dialect** defining the connection points for integrating 1. with 2.

To demonstrate the feasibility of their approach, a security modeling language for specifying access control requirements is defined. The language, called SecureUML, is embedded in an extension of UML. They give two examples of system design modeling languages: UML class diagrams and UML statecharts, and show how these languages can be combined with SecureUML.

For each combined modeling language, they define model transformations by augmenting model transformations for the UML-based modeling languages with the additional functionality necessary for translating their security modeling constructs. For the combination class diagrams + SecureUML, they define transformations to an access control infrastructure for distributed systems conforming to both the Enterprise JavaBeans (EJB) standard and the Microsoft Enterprise Services for .NET. For the combination statecharts + SecureUML, they define a transformation to access control infrastructures for web applications.

The work of Basin et.al. is both related to our method of model-driven information flow security and our method of model-driven policy specification. However, there are a number of differences. First, we do not consider access-control security requirements, but consider instead information flow security and prohibition policies. Second, we formally characterize the condition under which a transformation is security preserving w.r.t. secure information flow properties. This is in contrast to the work of Basin et. al. which does not provide a convincing argument of security preservation for the transformations they consider. The main reason for this is that no formal specification of the target platforms (EJB and .NET) is available. Without such a specification, proving security preservation is infeasible. Having said that, Basin et.al. do formally define what it means that a system (given as a labeled transition system) is secure w.r.t. a SecureUML model. However, the benefits of abstract analysis cannot be exploited unless the transformation to the concrete level is security preserving. A third difference between our work and the work of Basin et.al. is that they do not consider refinement and composition as we do.

Even though Basin et.al. do not give a formal proof about of security preservation of their transformations, we are not aware of any other approaches to MDS that does this either. In fact, the work of Basin et.al. constitutes the most comprehensive and precise approach to MDS that we are aware of.

### 7.5.3 Jan Jürjens: UMLSec

In [65–67], Jürjens considers the preservation of security under refinement. He also develops and extension of UML, called UMLSec, that allows for the specification of security relevant information (in the form of stereotypes and tagged values) in UML deployment diagrams, UML class diagrams, UML statecharts, UML activity diagrams, and UML sequence diagrams. Based on security relevant information in diagrams, he defines constraints that give criteria to evaluate the security aspects of a system design, by referring to a formal semantics of a simplified fragment of UML.
The work of Jürjens is related to ours in that he considers security, UML, and refinement in light of a formal foundation. However, there are many differences. Although he considers security requirements which we do not consider in this thesis, he does not consider security policies as we do. In addition, he only considers one particular secure information flow property, whereas we consider a more general security framework that can be instantiated to obtain specific secure information flow properties.

His approach to defining a notion of security preserving refinement is similar to ours. That is, like us, he distinguishes between two kinds of nondeterminism. The first kind corresponds to underspecification, leaving room for implementation decisions. This kind of nondeterminism may be eliminated in a refinement. The second kind of nondeterminism serves to protect secrets and is not meant to be eliminated in a refinement. Although Jürjens relies on this distinction to define a notion of secrecy preserving refinement. He does not rely on this distinction in the definition of his information flow property. Instead, he relies on a distinction between external nondeterminism (provided by the environment of a system) and internal nondeterminism (provided by the system) and effectively closes the information flow property under refinement of internal nondeterminism. This approach is less general than ours because it prohibits refinement of external nondeterminism, and it does not allow internal nondeterminism to be preserved under refinement.

The main difference between the work of Jürjens and our work, is that he does not consider transformations. Consequently his work cannot be said to be MDA compliant in the same way as ours.
Chapter 8

Conclusion

In this chapter, we conclude Part I. We first give a summary of what has been achieved, then we point to some directions of future work.

8.1 What has been Achieved

We have argued that security should be taken into account throughout the software development process, and that this may be achieved by integrating security into the MDA framework. MDA specialized to security is often called model driven security (MDS). Our goal has been to develop approaches to MDS that support

- (1) a stepwise modular development process in which transformations play an important role;
- (2) precise and sound foundation for rigorous security analysis;
- (3) preservation of security under refinement, transformation, and composition.

To achieve this goal, we have developed three artifacts:

(A) The formal foundation for model driven information flow security;

(B) The method for model driven information flow security;

(C) The method for model driven policy specification.

Artifact (A), the formal foundation for model-driven information flow security, consists of a security framework for specifying secure information flow properties that are preserved under refinement, and conditions under which security is preserved under transformation and composition. Artifact (A) is therefore an answer to goals (2) and (3). The advantage of the foundation is that it allows for a development process in which refinement can be used together with transformation and composition such that security is preserved. The only work that we are aware of that unifies the treatment of information flow security and the notions of refinement of underspecification, transformation, and composition is the work of Santen et.al. [52,106–108]. However, they address probabilistic security (as opposed to possibilistic security as we do) and their notion of data refinement is less general than our notion of transformation.

Artifact (B), the method for model driven information flow security, defines a process in which UML inspired state machines can be specified, shown to be secure w.r.t.
a set of security properties, and refined/transformed into more concrete state machines in such a way that security is preserved. Artifact (B) is therefore an answer to goal (1). We are not aware of any other approaches that exploit the full potential of MDS by supporting security analysis at a high level of abstraction whose validity can be preserved under transformation to a lower level of abstraction.

Artifact (C), the method for model driven policy specification, defines a process in which policies can be expresses in UML sequence diagrams and transformed into state machines that govern the behavior of execution monitoring mechanisms. The method supports the creation/selection of mapping rules which define how abstract behavior is transformed to concrete behavior. These mapping rules enable the specification of policies at a high-level of abstraction. Artifact (C) may be seen as an answer to goal (1). We are not aware of any related work in which UML sequence diagrams is used to specify policies in the setting of MDS.

### 8.2 Directions for Future Work

There are a number of possible directions of future work. As described in Sect.5.3.1, we would like to develop type checking rules which can be used by a tool to verify the security of specifications and the security preservation of transformations.

Initial work has already started on defining a transformation from state machines to source code. We believe that such a transformation would increase the pragmatic value of our method since most software development today involves imperative programming. One challenge then, is to maintain the relationship of the source code to the state machine it was generated from. Such a relationship is important for two reasons. First it could be used to ensure that as the source code evolves, it is still a refinement of the state machine it was generated from. Second, it could be used to generate a state machine from the source (reengineering) as the source code evolves.

A possible direction for the method of policy specification is to consider other enforcement techniques than execution monitoring. For instance, one could match the state machine generated from the policy against a target system at development time to check that the system adheres to the policy. The matching problem is in general undecidable, but we believe that approximation techniques could nevertheless yield useful results in many practical situations.
Bibliography


Part II

Research Papers
Chapter 9

Paper 1: Information Flow
Property Preserving
Transformation of UML Interaction
Diagrams
Information Flow Property Preserving Transformation of UML Interaction Diagrams

Fredrik Seehusen¹,² and Ketil Stolen¹,²
¹ SINTEF ICT, Norway
{Fredrik.Seehusen, Ketil.Stolen}@sintef.no
² Department of Informatics, University of Oslo, Norway

July 17, 2006

Abstract
We present an approach for secure information flow property preserving refinement and transformation of UML inspired interaction diagrams. The approach is formally underpinned by trace-semantics. The semantics is sufficiently expressive to distinguish underspecification from explicit nondeterminism. A running example is used to introduce the approach and to demonstrate that it is of practical value.

1 Introduction
Security is an important attribute of many software systems. Nevertheless, careful engineering of security into overall design is often neglected. Security features are typically built into an application in an ad-hoc manner or are only integrated during the final phases of system development [24]. It is, however, a common view in the security field that security mechanisms should be taken into account and built into the system during early phases of system development at a higher level of abstraction than the level of implementation.

The Object Management Group (OMG) advocates a framework for system development, Model Driven Architecture (MDA) [27], that aims to raise the level of abstraction of the programming environment by supporting (1) a model-driven development process, (2) a clear separation of abstract, platform independent models (PIMs) and refined, platform dependent models (PSMs), and (3) transformations between PIMs and PSMs.

By integrating security into MDA, security documentation can be specified and analyzed at a high level of abstraction during system development, thereby reducing the need for ad hoc integration of security mechanisms after system implementation. Moreover, the advantage of analyzing PIMs rather than PSMs is that analysis is more feasible at high level of abstractions as opposed to levels closer to implementation which may include too much detail to make analysis practical. Also, it is well known that the earlier an error is discovered, the easier and cheaper it is to fix.

In order to contribute towards a formal foundation of security within the MDA framework, we restrict the notion of security to a property of secure information flow. Secure information flow properties in general, provide a way
of specifying security requirements by selecting a set of domains, i.e. abstractions of real system entities such as users or files, and then restricting allowed flow of information between these domains. There are numerous definitions of information flow properties [25]. The property that we consider is based on a definition given in [20]. Informally, the property holds if an observer communicating with a system, based on its observations and its knowledge of the system specification, cannot infer that another observer has communicated with the same system.

The relationship between information flow security and refinement has been researched for a fairly long time. In 1989 it was shown by Jacob [20] that the derivation of secure systems from security specifications can be practically infeasible. It has later been observed e.g. in [17, 21, 30], that the problem originates in the inability of most specification languages, UML 2.0 [13] included, to distinguish between underspecification (potential nondeterminism) and explicit nondeterminism. An exception in this respect is STAIRS [14, 15, 16], an approach that provides a formal semantics for UML 2.0 inspired interaction diagrams where the distinction of potential and explicit nondeterminism is made. The approach presented in this report extends the theory of STAIRS wrt. refinement and information flow. It also provides the first steps towards a formal foundation for model-driven security in the setting of UML interactions. In particular, the contributions of this report are:

- A formal definition of a notion of secure information flow in STAIRS. Thereby we provide a foundation for analysis of UML 2.0 interaction diagrams wrt. secure information flow. The only work that we are aware of that deals with information flow security in a proposed semantics of UML is [21], but only a simplified version of UML 1.5 [12] interaction diagrams is considered in this work.

- Showing that the STAIRS notion of refinement preserves our notion of secure information flow. The most notable recent theoretical works that we are aware of that address refinement and secure information flow are [5, 26]. In [5], conditions for checking that a given refinement is information flow preserving are presented, and [26] proposes a way of modifying refinement operations such that they remain secure information flow preserving. These approaches are clearly different from ours. We prove that all refinements preserve our notion of secure information flow, thus there is no need to check or modify given refinements as in [5, 26].

- A formal definition of a notion of transformation that preserves our notion of secure information flow. The notion of refinement is a relation between two specifications and it is in general not subject for automation. In contrast, a transformation takes a specification and delivers another specification as output in the sense of a compiler. There are no related works that we are aware of that address secure information flow with respect to this notion of transformation.

- A generalization of the notion of refinement into a more general notion of refinement modulo transformation. This more general concept may be

1 Also termed unpredictability [21] and probabilistic nondeterminism [30].
The report is structured as follows: In Sect. 2 we use STAIRS to specify a PIM of an example system which is sufficiently realistic to suggest that our approach is of practical value. In Sect. 3 we define an information flow property in terms of the STAIRS semantics, and explain why our PIM is secure with respect to this property. Sect. 4 presents a formalization of refinement in the STAIRS semantics, and explains why the information flow property is preserved under refinement. In Sect. 5 we introduce a notion of transformation which is information flow property preserving. We also integrate this notion of transformation and the classical notion of refinement from STAIRS into a more general concept. In Sect. 6 we show how our PIM can be transformed to PSMs in such a way that the information flow property is preserved. Sect. 7 describes related work. Sect. 8 provides conclusions and suggests directions for future work. The appendix provides formal definitions of auxiliary operators used in this report.

2 The Project Management System

Consider a large software developing company that aims to develop a distributed system, the project management system (the PM system), in order to centralize all storage of software development projects. Software developers should be able to retrieve projects from a server to their local machines, edit or add files to the project, and upload any changes back to the server. Projects stored on the server should be versioned, and whenever a developer updates a project, only changes in the project with respect to the developer’s local copy should be updated.

Currently, the company has no unified development method, and developers working on different projects are to a large degree given flexibility in the method they choose to adopt. The company wants to assess the different methods in order to recommend improvements, and possibly to introduce a unified development process. This task is assigned to a group of researches. The researchers are to pick a set of sample projects, and assess each project thoroughly with respect to progress, quality of code etc. For convenience, the PM system should be augmented slightly such that the researchers will be able to retrieve projects from the server over the Internet on a regular basis. This additional functionality is not of high priority, thus it will be implemented with little resources. The researcher will not use the same versioning system that the developers are understood as a kind of data refinement [19].

Figure 1: Overview of the Project Management System
using, thus for simplicity, the whole project is copied to the researchers for every retrieval.

To make the assessment as realistic as possible, the software developers of the company should not know which projects are sampled for the assessment. We ensure this by requiring that the developers should not, at any given moment in time, be able to deduce whether a researcher has retrieved a project from the server. This means that the developers are ignorant of the researchers.

The company decides that the server where all projects are stored shall expose two web-service endpoints, one for the developers and one for the researchers as illustrated in Fig. 1. Also, the server implementation will be based on J2EE technology, where so-called Java session beans are responsible for handling SOAP-message communication between the server and the clients, and the Java entity beans are responsible for handling persistence.

2.1 Capturing Behavior Using STAIRS

The interaction diagrams in Fig. 2 specify two simple interactions of the PM system in which a developer updates a project by sending an update message to the server, whereupon to server responds by returning an ok or an error message depending on whether or not the update was successful.

In the graphical diagrams, vertical dashed lines correspond to so-called lifelines, and the signals of messages correspond to the labels decorating the arrows between the lifelines.

Definition 1 A message is a triple \((tr, re, si)\), consisting of a transmitter \(tr\), a receiver \(re\), and a signal \(si\).

Example The two messages in the left most diagram of Fig. 2 are represented by the triples \((D, S, u)\) and \((S, D, o)\). For conciseness we refer to the lifelines and the signals in graphical diagrams by their first letters if this can be done unambiguously.

In the trace-semantics of STAIRS [15], the transmission and the reception of a message \(m\) is represented by a transmission event (denoted \((!, m)\)) and a reception event (denoted \((?, m)\)), respectively. For conciseness, an event \((!, (tr, re, si))\) is usually denoted \(!si\) if the transmitter and the receiver can be deduced from the context. The same convention applies for reception events.

A trace is a sequence of events representing a single run. A trace is causal and weakly sequenced. Causality means that a message must be transmitted before it is received. Weak sequencing [13] means that events are ordered by their vertical position with respect to each lifeline. For example, if two messages
are transmitted from a lifeline $l_1$ to a lifeline $l_2$, then the first message is not necessarily received by $l_2$ before the second message is transmitted from $l_1$.

**Definition 2** An event is a pair, $(k, m)$, consisting of a kind $k \in \{!, ?\}$ and a message $m$. A trace is a sequence of events. We let $\mathcal{H}$ denote the set of all traces.

**Example** The diagrams PM\(_1\) and PM\(_2\) in Fig. 2 describe the traces $\langle !u, ?u, !o, ?o \rangle$ and $\langle !u, ?u, !e, ?e \rangle$, respectively.

STAIRS distinguishes between positive and negative traces. Positive traces represent desired or acceptable behavior, while negative traces represent undesired or unacceptable behavior. The remaining traces are inconclusive meaning that their status is not decided or that they are irrelevant. The semantics of a diagram in STAIRS is set of pairs of positive and negative traces. Such pairs are referred to as interaction obligations. Each interaction obligation represents an explicit nondeterministic alternative. Underspecification (also referred to as potential nondeterminism) is represented by allowing the implementer to choose between different alternative behaviors within a single interaction obligation.

**Definition 3** An interaction obligation is a pair, $(p, n)$, where $p$ is a set of positive traces and $n$ is a set of negative traces. The semantics of a diagram $d$ in STAIRS, written $\llbracket d \rrbracket$, is a set of interaction obligations.

Unless otherwise indicated, traces of interaction diagrams are interpreted as positive.

**Example** Diagram PM\(_1\) and PM\(_2\) of Fig. 2 describe the interaction obligations $\langle \{\langle !u, ?u, !o, ?o \rangle\}, \emptyset \rangle$ and $\langle \{\langle !u, ?u, !e, ?e \rangle\}, \emptyset \rangle$, respectively. The set of negative traces in each interaction obligation is empty since traces of diagrams are interpreted as positive by default.

### 2.2 Potential Nondeterminism

The diagram in Fig. 3 specifies the interaction in which a developer tries to update a project before it is retrieved by a researcher who has requested the same project prior to the developer’s update attempt. The updated project cannot be stored on the server while it is being read, but whether or not the update will have to be resubmitted (upon reception of an error message) by the developer is a postponed design decision.
In interaction diagrams as interpreted in STAIRS, underspecification in the form of alternative design decisions is explicitly expressed by the so-called alt-operator as illustrated in Fig. 3. Interactions that are separated by the alt-operator are interpreted as potential nondeterministic trace alternatives. In Fig. 3 the topmost operand of alt specifies the design alternative in which an ok message is transmitted back to the developer. The second alternative specifies the alternative where the developer receives an error message.

**Definition 4** The alt-operator specifies potential nondeterminism by leaving the choice between its operands open. The semantics is the inner union of each point-wise selection of interaction obligations from its operands:

\[
\left[ \text{alt}[d_1, \ldots, d_n] \right] \overset{\text{def}}{=} \{ \bigcup_{\forall i \in [1,n]} o_i \mid \forall i \in [1,n] : o_i \in [d_i] \}
\]

The inner union of interaction obligations is defined as:

\[
\bigcup_{i \in [1,n]} (p_i, n_i) \overset{\text{def}}{=} \bigcup_{i \in [1,n]} p_i, \bigcup_{i \in [1,n]} n_i
\]

**Example** The diagram PM3 in Fig. 3 describes an interaction obligation \((p, \emptyset)\) where \(p\) includes the traces \(\langle g, g, u, a, o, !r, ?r \rangle\) and \(\langle g, g, u, ?u, c, ?e, !r, ?r \rangle\) due to the alt-operator. Note that \(p\) will also contain other traces due to weak sequencing. E.g. \(p\) will also contain the trace \(\langle g, u, g, ?u, c, ?e, !r, ?r \rangle\), where both the messages get project (!\(g\)) and the update project (!\(u\)) are transmitted by the researcher and the developer, respectively, before the corresponding messages (?\(g\) and ?\(u\)) are received by the server.

### 2.3 Explicit Nondeterminism

In STAIRS we may use the xalt-operator to specify explicit nondeterminism. This is illustrated in the interaction diagram of Fig. 4. It states that any correct implementation must offer the choice between the operands PM1, PM2, and PM3. Each alternative specified by an xalt-operator represents an explicit nondeterministic choice that must be preserved by any correct refinement of the diagram.
Definition 5 The \texttt{xalt}-operator specifies explicit nondeterminism by requiring all operands to be reflected in any proper implementation. The semantics is the union of the sets of interaction obligations characterized by the operands:

\[
\llbracket \texttt{xalt}[d_1, \ldots, d_n] \rrbracket \equiv \bigcup_{i \in [1..n]} \llbracket d_i \rrbracket
\]

Assume now for the sake of the running example, that the diagram in Fig. 4 describes all the relevant interactions of the PM system. We may then encapsulate the interactions of the diagram using the \texttt{assert}-operator (as shown in Fig. 4), which means that all traces that are not described within the scope of the assert, i.e. the inconclusive traces, are to be interpreted as negative. This is of course an incomplete specification, but interaction diagrams are often used to capture incomplete aspects of a system.

Definition 6 The \texttt{assert}-operator makes all inconclusive traces negative. The sets of positive and negative traces are left unchanged:

\[
\llbracket \texttt{assert } d \rrbracket \equiv \{(p, n \cup (\mathcal{H} \setminus p)) \mid (p, n) \in [d]\}
\]

Example The STAIRS-semantics interprets the diagram PM of Fig. 4 as \{(p_1, \mathcal{H} \setminus p_1), (p_2, \mathcal{H} \setminus p_2), (p_3, \mathcal{H} \setminus p_3)\} where the trace sets \(p_1, p_2, \) and \(p_3\) are described by the diagrams PM\(_1\), PM\(_2\), and PM\(_3\), respectively.

A diagram is well-defined if its semantic interpretation does not contain an interaction obligation \((p, n)\) such that \(p \cap n \neq \emptyset\). We denote by \(\mathcal{D}\) the set of all well-defined diagrams that can be constructed by the syntactic operators presented in this report.

3 Why Our Specification Ensures Ignorance

A requirement of the PM system is that the developers should be ignorant of the researchers. In this section we formalize this notion. For this purpose, we distinguish between observers and the system under observation. Each observer is only able to observe its own interaction with the system, and we assume that each observer has complete knowledge of the diagram describing the system they interact with. Based on this knowledge, an observer can construct the set of all system behaviors which are compatible with a given observation, the so-called inference set, and try to deduce confidential information from this set. The secure information flow property that we formalize (the ignorance property) demands that an observer \(o\) must not, based on the inferences it can make, deduce with certainty that another observer \(o'\) has communicated with the system under observation.

3.1 Systems, Behavior and Observations

In this section we define the notions of system, observer, behavior, and observation in terms of the STAIRS semantics.

Definition 7 Systems and observers are represented by sets of lifelines.
Example In our running example, we require that developers should be ignorant of researchers. Thus developers and researchers are observers. With respect to the diagram PM (of Fig. 4), let $D$ and $R$ denote the observers $\{\text{: Developer}\}$ and $\{\text{: Researcher}\}$, respectively. Both the developer and the researcher communicate with the same server. Hence, the server is the system under observation. We let $S$ denote the system $\{\text{: Server}\}$. This applies throughout the rest of our examples.

STAIRS does not prohibit an implementation of a diagram to produce traces that are not described by the diagram (the inconclusive traces). The set of traces allowed by an interaction obligation is therefore its set of non-negative traces (i.e. the union of its positive and inconclusive traces). Such traces are called behavior traces. The set of all behavior traces that are derived from exactly one interaction obligation is called a behavior alternative.

In general, an interaction diagram may describe the behavior of several systems or observers. We are usually interested in reasoning about the behavior of a specific system. A behavior alternative of a system $s$ can be obtained from an interaction obligation $a$ by restricting all behavior traces in $a$ to the lifelines in $s$.

Definition 8 The behavior alternative of system $s$ in interaction obligation $(p,n)$, written $\nabla((p,n), s)$, is defined:

$$\nabla((p,n), s) \overset{\text{def}}{=} \{e.s \circ h \mid h \in H \setminus n\}$$

Here $e.s$ is the set of events that may occur on the lifelines in $s$. Furthermore, the function $A \circ h$ yields the trace obtained from trace $h$ by filtering away all events in $h$ that are not in the set of events $A$ (see App. A).

We lift the function to yield the behavior alternatives of a system $s$ as described in a diagram $d$.

Definition 9 The behavior of a system $s$ in diagram $d$, written $\nabla(d,s)$, is defined:

$$\nabla(d,s) \overset{\text{def}}{=} \bigcup_{a \in \mathcal{I}(d)} \{\nabla(a,s)\}$$

Example Let PM be the diagram specified in Fig. 4. The behavior of system $S$ in PM is given by the behavior traces that can occur on the lifeline in $S$: $\nabla(PM, S) = \{b_1, b_2, b_3\}$, where $b_1 = \{(?u, !o)\}$ (corresponding to PM$_1$ in Fig. 2), $b_2 = \{(?u, !e)\}$ (corresponding to PM$_2$ in Fig. 2), and $b_3 = \{(?y, ?u, !o, !r), (?r, ?u, !e, !r)\}$ (corresponding to PM$_3$ in Fig. 3).

An observer may observe its own interaction with the system under observation. The set of observation traces that an observer can make of a behavior alternative $b$ is obtained from $b$ by filtering away the events that are not transmitted to or from the observer.

Definition 10 The set of all observation traces that an observer $o$ can make of behavior alternative $b$, written $o \triangleright b$, is defined:

$$o \triangleright b \overset{\text{def}}{=} \{\text{rt}.o \circ h \mid h \in b\}$$
rt.o yields the set of events that can be transmitted to or from the observer o (see App. A).

We lift the observation function to diagram-system pairs.

**Definition 11** The set of all observation traces that an observer o can make of system s in diagram d, written o ⪰ (d, s), is defined:

\[ o ⪰ (d, s) \equiv \bigcup_{h \in \nabla(d,s)} o \equiv b \]

The definition implies that an observer may observe liveness properties (such as termination) because neither the set of behavior traces nor the set of observation traces have to be prefix-closed [3].

**Example** Based on Fig. 4, we have that \( D \triangleright (PM, S) \) yields the observation set \( \{\langle ?u, !o \rangle, \langle ?u, !e \rangle\} \), and \( R \triangleright (PM, S) \) yields the set \( \{\langle ?g, !r \rangle\} \).

### 3.2 Inference and Ignorance

We assume that observers have full access to the specification of the system, i.e., the observers have complete knowledge of the behavior alternatives of the system under observation. An observer may then, based on a behavior alternative \( b \), construct an inference set consisting of all traces in \( b \) that are compatible with a given observation \( h \).

**Definition 12** The inference set that observer \( o \) can construct from behavior alternative \( b \), based on observation trace \( h \), written \( o \triangleright \! b \), is defined:

\[ o \triangleright \! b \equiv \{b' \in b \mid h = (rt.o \otimes b')\} \]

**Example** Continuing the previous examples, let \( h_1 \) and \( h_2 \) be observation traces \( \langle ?u, !o \rangle \) and \( \langle ?u, !e \rangle \) of observer \( D \), respectively, and let behavior alternative \( b \) be the set \( \{\langle ?g, ?u, !o, !r \rangle, \langle ?g, ?u, !e, !r \rangle\} \). We have that observer \( D \), based on \( h_1 \), can infer the following set of behavior traces from \( h_1 \): \( D \triangleright \! b = \{\langle ?g, ?u, !o, !r \rangle\} \). Similarly, we have that \( D \triangleright \! b_2 = \{\langle ?g, ?u, !e, !r \rangle\} \).

If an inference set that an observer \( o \) can construct has a trace that compromises another observer \( o' \), meaning that the trace contains an event that is sent to or received from \( o' \), then \( o \) may deduce that \( o' \) has done something. Conversely, if none of the traces in the inference set compromise \( o' \), then \( o \) cannot deduce that \( o' \) has done something. We say that a behavior alternative from which such an inference set can be constructed hides \( o' \).

**Definition 13** Behavior alternative \( b \) hides observer \( o' \) from observer \( o \) wrt. observation \( h \), written \( o \triangleright \! b \triangleright \! o' \), iff

\[ \forall b^h \in o \triangleright \! b : rt.o' \otimes h^b = \langle \rangle \]

**Example** Consider diagram PM_3 (Fig. 3). Let \( b \in \nabla(PM_3, S) \) and \( h \in D \triangleright (PM_3, S) \), such that behavior alternative \( b \) equals \( \{\langle ?g, ?u, !o, !r \rangle, \langle ?g, ?u, !e, !r \rangle\} \) and observation \( h = \langle ?u, !o \rangle \). The set of traces that \( D \) can infer from \( b \) based on \( h \) equals \( D \triangleright \! b = \{h^b\} \) (where \( h^b = \langle ?g, ?u, !o, !r \rangle \)). Since the trace \( h^b \) contains events that compromises observer \( R \) \( \langle rt.R \otimes h^b = \langle ?g, !r \rangle \rangle \), we have that \( b \) does
not hide \( R \) from \( D \) wrt. \( h \), i.e. \( D \not{\bowtie}_h R \) yields false.

Observer \( o \) is ignorant of observer \( o' \) if \( o \) cannot infer with certainty that \( o' \) has communicated with the system under observation. More precisely, this means that if \( o \), based on an observation \( h \) and a behavior alternative \( b_1 \), is able to infer a trace that compromises \( o' \), then there must exist another behavior alternative \( b_2 \) that hides \( o' \) from \( o \) wrt. the same observation \( h \).

**Definition 14** An observer \( o \) is ignorant of observer \( o' \) wrt. diagram-system pair \( S_d \), written \( o \not{\bowtie}_S o' \), iff

\[
\forall h \in o \triangleright S_d : \forall b_1 \in \nabla S_d : \neg(o \not{\bowtie}_h b_1) \Rightarrow \exists b_2 \in \nabla S_d : o \not{\bowtie}_h o' \wedge b_2 = o <_h b_2
\]

The last condition (\( b_2 = o <_h b_2 \)) ensures that the ignorance property is preserved during refinement. This will be explained in Sect 4.

**Example** \( D \) is ignorant of \( R \) wrt. \( S \) in diagram PM (Fig. 4). To see this, note that for each observation trace \( h \) that \( D \) can make of \( S \), we have that for all traces that compromise \( R \) that \( D \) can infer based on \( h \), there exists a behavior alternative that hides \( R \) from \( D \) wrt. \( h \). E.g. as described in the previous example, \( D \) can infer the compromising trace \( \langle ?g, ?u, !o, !r \rangle \) based on observation \( h = \langle ?u, !o \rangle \). However, there also exists a behavior alternative \( b \in \nabla (PM, S) \) that hides \( R \) wrt. \( h \). Specifically, \( b = \{\langle ?u, !o \rangle\} \) (corresponding to the interaction obligation described by diagram PM_1 in Fig. 2). We have that \( D \), based on \( h \), can infer the trace \( \langle ?u, !o \rangle \) from \( b \). Since this trace does not compromise \( R \) \((\forall t. !R \circ \langle ?u, !o \rangle = \langle \rangle)\), all traces in \( b \) (since there is only one trace in \( b \)) hide \( R \) from \( D \) wrt. \( h \).

### 4 Why the Ignorance Property is Preserved by Refinement

Refinement means to add information to a specification such that the specification becomes more complete. This may be achieved by categorizing inconclusive traces as either positive or negative, or by reducing the set of positive traces. Negative traces always remain negative. We first define refinement of interaction obligations formally. This definition is then lifted to interaction diagrams.

**Definition 15** An interaction obligation \((p_c, n_c)\) is a refinement of an interaction obligation \((p_a, n_a)\), written \((p_a, n_a) \leadsto (p_c, n_c)\), iff

\[
n_a \subseteq n_c \land p_a \subseteq p_c \cup n_c
\]
Definition 16 An interaction diagram \( d_c \) is a refinement of an interaction diagram \( d_a \), written \( d_a \rightarrow d_c \), iff

\[
(\forall a \in \llbracket d_a \rrbracket : \exists c \in \llbracket d_c \rrbracket : a \rightarrow c) \land (\forall c \in \llbracket d_c \rrbracket : \exists a \in \llbracket d_a \rrbracket : a \rightarrow c)
\]

Example The diagram \( PM_3R \) (Fig. 5) is a refinement of diagram \( PM_3 \) (in Fig. 3). That is, \( \llbracket PM_3 \rrbracket = \{ (p_a, \emptyset) \} \) and \( \llbracket PM_3R \rrbracket = \{ (p_c, H \backslash p_c) \} \) such that \( p_c \subseteq p_a \). \( PM_3 \) differs from \( PM_3R \) in that some positive traces in \( PM_3 \) and all inconclusive traces have been made negative in \( PM_3R \).

Theorem 1 The ignorance property is preserved by refinement, formally:

\[
\sigma (d_{(s,a)}) o' \land d_a \rightarrow d_c \Rightarrow \sigma (d_{(s,a)}) o'
\]

The proof of the theorem relies on the fact that the ignorance property is defined in terms of behavior alternatives as opposed to individual traces which is how secure information flow properties are usually defined in the literature [25]. That is, instead of demanding that there exists a trace that fulfills a certain criterion, we demand that there exists a behavior alternative such that all its traces fulfills a certain criterion. In Def. 14, this criterion is expressed by the condition \( o_0 \bigcirc_h b_{2} \), which requires that none of the traces in \( b_{2} \) that are compatible with observation \( h \) must compromise observer \( o' \). The condition \( b_{2} = o \bigcirc_h b_{2} \) demands that all traces in \( b_{2} \) must be compatible with observation \( h \). This ensures that all traces in \( b_{2} \) hide \( o' \) from \( o \). Since all (not just some) traces in \( b_{2} \) must fulfill this requirement, the criterion holds regardless of how \( b_{2} \) is refined. Note that we do not have to worry about empty behavior alternatives, since no well-defined diagram can yield such a behavior alternative given the syntactic constructs presented in this report.

5 Ignorance Preserving Transformation

The notion of refinement addressed above is a binary relation on specifications that formalizes the process of stepwise development by removal of underspecification. This process depends on human intuition and is in general not subject for automation. A transformation on the other hand, is an executable function that takes a syntactic specification and produces another specification (e.g. a PIM to a PSM). Semantically, we represent a syntactic transformation by a set of functions mapping traces over abstract events to traces over concrete events. The reason why a single syntactic transformation is represented by a set of functions is that transformations often introduce a finer granularity in the sense that one abstract trace may correspond to a set of concrete traces. Alternatively, we could have used a function mapping traces to sets of traces.

5.1 Transformations from a Semantic Perspective

We denote by \( H_a \) a set of traces over an abstract set of events, and by \( H_c \) a set of traces over a concrete set of events. Hence, contrary to earlier, we now have an abstract universe \( H_a \) and a concrete universe \( H_c \). Transformations are semantically represented by a set of functions

\[
F \subseteq H_a \rightarrow H_c
\]
that maps abstract traces to concrete traces.

In order to ensure that transformations preserve our security property, we impose two requirements on each function \( f \in F \).

First, we require \( f \) to be transmitter and receiver preserving in the following sense:

\[
\text{condition (1)}
\]

Here \( h_a \) is universally quantified over \( H_a \). Similarly, \( l_1 \) and \( l_2 \) range over the set of all lifelines. The function \( e_{l_1,l_2} \) yields the set of events that can occur on lifeline \( l_1 \) whose message is sent from \( l_1 \) to \( l_2 \) or from \( l_2 \) to \( l_1 \).

Condition (1) states that an abstract trace whose events all have the same transmitter and receiver, say \( t \) and \( r \), must by transformed to a concrete trace whose events also have the same transmitter and receiver \( t \) and \( r \). Note that condition (1) may be weakened; e.g. by allowing lifelines to be decomposed into a set of lifelines.

The second condition requires each \( f \in F \) to be homomorphic wrt. concatenation of traces:

\[
\text{condition (2)}
\]

The traces \( h_1 \) and \( h_2 \) are universally quantified over \( H_a \). Condition (2) essentially states that \( f \) is a function of events rather than trace histories.

Note that requirements (1) and (2) ensure that one abstract observation is transformed to one and only one concrete observation. We let \( F \) denote the set of all functions that abide to requirements (1) and (2) above.

### 5.2 Ignorance Preserving Transformation of Interaction Diagrams

In the following, \( d_a \) and \( d_c \) are interactions diagrams with traces in \( H_a \) and \( H_c \), respectively. Moreover, \( T \) is a (syntactic) transformation on interaction diagrams whose semantics \( \llbracket T \rrbracket \subseteq F \) is a transformation as defined above.

We first define what it means to translate an interaction obligation. Then we lift this notion of translation to interactions diagrams.

**Definition 17** Interaction obligation \((p_c,n_c)\) is a translation of interaction obligation \((p_a,n_a)\) with respect to function \( f \in F \), written \((p_a,n_a) \mapsto f \ (p_c,n_c)\), iff

\[
\mathcal{H}_c \setminus n_c = \{ f(h_a) \mid h_a \in (\mathcal{H}_a \setminus n_a) \} \land p_c = \{ f(h) \mid h \in p_a \}
\]

The first condition states that each non-negative trace of the concrete interaction obligation must be a translation of a non-negative trace of the abstract obligation. The second condition ensures that a positive trace in the abstract
obligation is transformed to a positive (not inconclusive) trace in the concrete obligation.

**Definition 18** An interaction diagram $d_c$ is a translation of an interaction diagram $d_a$ with respect to transformation $T$, written, $d_a \hookrightarrow_{T} d_c$, iff

$$\llbracket d_c \rrbracket = \{ c | a \hookrightarrow f c \land a \in \llbracket d_a \rrbracket \land f \in \llbracket T \rrbracket \}$$

If $F_1, F_2 \subseteq \mathcal{F}$ then $F_1 \circ F_2$ is understood as the set of functions obtained by functional point-to-point composition of all functions in $F_1$ and $F_2$. That is,

$$F_1 \circ F_2 \overset{\text{def}}{=} \{ f_1 \circ f_2 | f_1 \in F_1 \land f_2 \in F_2 \}$$

If $T_1$ and $T_2$ are syntactic transformations, we write $T_1 \circ T_2$ instead of $\llbracket T_1 \circ T_2 \rrbracket$.

**Lemma 1** $\hookrightarrow$ is transitive, formally:

$$d_1 \hookrightarrow_{T_1} d_2 \land d_2 \hookrightarrow_{T_2} d_3 \Rightarrow d_1 \hookrightarrow_{T_2 \circ T_1} d_3$$

**Theorem 2** The ignorance property is preserved by transformation, formally

$$o^{\{d_a, s\}} o' \land d_a \hookrightarrow_{T} d_c \Rightarrow o^{\{d_c, s\}} o'$$

The proof relies on two facts. First, all transformations are interpreted by a set of functions where each function by definition abides to conditions (1) and (2) that together ensure that one observation at the abstract level is transformed to one and only one observation at the concrete level. Second, each function that interprets a transformation is by Def. 18 applied to interaction obligations in the manner illustrated in Fig 6. That is, each function translates each interaction obligation in the abstract diagram into a concrete interaction obligation. This ensures that additional granularity introduced at the concrete level is in the form of explicit nondeterminism as opposed to potential nondeterminism because the cardinality of an obligation (i.e. the number of traces in the obligation that may provide potential nondeterminism) is never increased during transformation.

The notions of refinement and transformation may be combined into a more general notion of refinement.

**Definition 19** The interaction obligations $o_a$ and $o'_c$ are in a refinement relation wrt. function $f \in \mathcal{F}$, written $o_a \sim_f o'_c$, iff

$$\exists o_c \in \mathcal{O} : o_a \sim_f o_c \land o_c \sim o'_c$$

**Definition 20** Two interaction diagrams $d_a$ and $d'_c$ are in a refinement relation wrt. transformation $T$, written $d_a \sim_T d'_c$, iff

$$\llbracket d'_c \rrbracket = \{ d'_c \in \mathcal{O} | o_a \sim_f o'_c \land o_a \in \llbracket d_a \rrbracket \land f \in \llbracket T \rrbracket \}$$

**Lemma 2** $\sim$ is transitive modulo translation.

$$d_1 \sim_{T_1} d_2 \land d_2 \sim_{T_2} d_3 \Rightarrow d_1 \sim_{T_2 \circ T_1} d_3$$

The results on preservation of the ignorance property carry over to the general notion of refinement.

**Corollary 1** The ignorance property is preserved by refinement modulo transformation, formally:

$$o^{\{d_a, s\}} o' \land d_a \sim_T d_c \Rightarrow o^{\{d_c, s\}} o'$$

The proof is straightforward given Theorem 1 and 2.
6 Transforming the Project Management System

In Sect. 5 we showed that our notion of transformation preserves the ignorance property. The purpose of this section is to show that our notion of transformation is not so strong that it cannot be used to semantically characterize transformations of practical value.

In our running example, the developers and the researchers of the PM system communicate with the server by SOAP messages. SOAP, however, is typically bound to HTTP which again typically runs over TCP, etc. In the following we demonstrate how the PM system based on SOAP can be transformed into a system based on TCP in such a way that the ignorance property is guaranteed to be preserved even when developers are able to observe TCP messages as opposed to SOAP messages. In MDA terminology, this can be seen as a transformation from a PIM to a PSM, where the platform is understood as a protocol.

6.1 SOAP to TCP

In practice, a transformation from SOAP to TCP typically works as follows: First each SOAP request and response message is encapsulated by a HTTP request and response header, respectively, then a new so-called TCP session is created for each HTTP request-response pair. That is, for each HTTP request-response pair, (1) a connection is established between the two sides of communication, (2) both the request and response are segmented, encapsulated by TCP frames and transmitted, (3) finally the connection is explicitly terminated. The TCP protocol must, among other things, handle message overtaking and message disappearance. Hence, a HTTP request-response pair may be translated into one of several potential TCP session alternatives.

Let $d_s \rightsquigarrow_s 2h \rightsquigarrow_h 2t \rightsquigarrow_t d_t$ be two transformations that translate diagrams at the SOAP level to diagrams at the HTTP and TCP levels. $s2h$ may be interpreted by the set $\llbracket s2h \rrbracket$ consisting of one bijective function that substitutes the signal $s_i$ of each event in $d_s$ with a signal that represents its encapsulated HTTP equivalent, e.g. by concatenating the letter “H” and $s_i$.

$H2t$ can be interpreted
as the set \([ h2t ]\) such that each function in \([ h2t ]\) translates a HTTP request message and a HTTP response message into a TCP session alternative. The fact that the functions in \([ s2h ]\) and \([ h2t ]\) satisfy requirements (1) and (2) is almost immediate. (1) is satisfied because the transformation preserves the transmitters and receivers of events. (2) is satisfied because \( s2h \) and \( h2t \) can be expressed as a function of single events (as opposed to traces). Since translations are transitive by Lemma 2, \( d_s \xrightarrow{s2h} d_h \xrightarrow{h2t} d_t \) is ignorance preserving by Theorem 2.

**Example** The two left most diagrams in Fig. 7 illustrate the case where \( s2h \) is applied to diagram PM\(_1\) (also specified in Fig. 2). Here, the SOAP level message names \( update(p) \) and \( ok \) have been transformed to the HTTP level message names \( Hupdate(p) \) and \( Hok \) in diagram PM\(_{1h}\). Diagrams PM\(_{1h}\) and PM\(_{1t}\) illustrate the application of \( h2t \) where the HTTP request and response messages are translated in a number of different TCP session alternatives. These alternatives are composed by the \texttt{xalt}, since by our definition of transformation, each function in \([ h2t ]\) yields an explicit nondeterministic alternative.

## 7 Related Work

The work that is most related to ours is the work of Jürgens, largely summarized in the book [21]. Not only does he give a formal semantics for a fragment of UML, but he also identifies conditions under which a secure information flow property (among other properties) is preserved under refinement. There are appreciable differences between our work and his. First, he formalizes a fragment of UML in which only a simplified version of sequence diagrams of UML 1.5 is included. In other words, while his work is based on formalized parts of UML that we do not consider, our work is based on STAIRS which provides a more in depth formalization of UML 2.0 interactions than what is considered in Jürgens work. Second, his semantics is based on so-called UML machines which are a kind of state machines. The semantics of STAIRS is not based on state machines. Third, while Jürgens distinguishes between potential and explicit nondeterminism in order to define a refinement preserving property of confidentiality and integrity, he does not rely on this distinction in the definition of his information flow property. That is, a secure information flow property for UML machines is formalized such that each behavior refinement to a deterministic UML machine satisfies this property, i.e. he effectively closes the property under refinement without considering explicit nondeterminism. He himself notes this is “rather strong” and it is in fact exactly this kind of definition that we have been trying to avoid. Our experience suggests that information flow properties closed under refinement tend to be too strong for practical use if the distinction between potential and explicit nondeterminism is not taken into consideration. For example, if we had based our running example on a definition of ignorance similar to that which is given in [20] (where explicit nondeterminism is not considered), and then closed this property under refinement without considering explicit nondeterminism, then the property would have been so strong that the example would not have made any sense. Finally, Jürgens does not address MDA or transformations.

Other efforts that are comparable to ours on a general level in that they
address security in an MDA-setting and/or model-based security, can be roughly classified into (1) access control related works: [2, 4, 7, 8, 9, 18, 22, 24, 28, 29], (2) secure database development [10], and (3) specification of high-level security requirements [1, 6, 11, 31]. Although all these works are comparable to ours on a general level, they are in fact quite different in that they do not deal with a formalized notion of refinement, nor do they address information flow security.

There are a number of more theoretical papers related to information flow security and refinement. The work of Jacob [20] is related to ours in that our formalization of ignorance is based on Jacob’s formalization of this property. Jacob’s property is, however, not preserved under refinement. In fact, he was the first that we are aware of to show that the derivation of secure systems from security specifications can be practically infeasible. This became known as the refinement paradox. It has later been observed that this “paradox” is a manifestation of failing to clearly distinguish between potential and explicit nondeterminism. As far as we are aware of, this observation was first made in [30].

Two notable recent theoretic works that define refinement operators that preserve information flow properties under refinement are [5, 26]. Both papers investigate a number of information flow properties, but their approaches differ from ours. Specifically, [5] presents sufficient conditions with which to check that a given refinement (defined in terms of simulation in a processes calculus) preserves information flow properties. Similarly, [26] presents refinement operators that can be used to check or modify refinements such that security is preserved. Both papers differ from ours in that we address secure information flow preservation in a formalism that distinguishes between potential and explicit nondeterminism. By taking this distinction into consideration in our definition of refinement and the ignorance property, we show that all refinements preserve ignorance, thus there is no need to check each specific refinement.

The work of Heisel. et. al. [17] is similar to ours in that they distinguish between potential and explicit nondeterminism. The main differences between their work and ours are: (1) They work in a probabilistic setting in a different formalism than ours and they do not consider UML interaction diagrams. (2) They consider a notion of confidentiality based on low-level indistinguishability. This notion of security is strictly speaking not a secure information flow property. (3) Their approach is different from ours in that they build the condition of security preservation into their notion of confidentiality preserving refinement. Wrt. refinement, we take the dual approach of strengthening the notion of security, i.e. by strengthening the ignorance property wrt. its original proposal [20].

8 Conclusions and Future Work

On a general level, we have argued that by integrating security into MDA, security documentation can be specified and analyzed at a high level of abstraction during early phases of system development, thereby reducing the need of ad hoc integration of security mechanisms at the level of implementation. Moreover, analysis is more feasible at high levels of abstraction than at levels closer to implementation because specifications at these levels may be too detailed to make analysis practical.
Our main objective has been to contribute towards a formal foundation for this by presenting an approach where high-level UML inspired interaction diagrams can be analyzed with respect to a secure information flow property, and transformed, if desired, down to the level of sequences of bits in such a way that the property is preserved.

Our approach is based on STAIRS [16], and the contributions of this report are specifically (1) the formalization of the ignorance property (Sect. 3), (2) showing that the ignorance property is preserved by the STAIRS notion of refinement (Sect. 4), and (3) the formalization of ignorance preserving transformations and the general notion of refinement (Sect. 5). We are not aware of any other work that exploits the distinction of potential and explicit nondeterminism in order to define a notion refinement in such a way that all refinements preserve a secure information flow property. Moreover, we are not aware of any work that addresses secure information flow and a notion of transformation.

A natural direction of future work is to generalize our results to other information flow properties. We are also planning to consider syntactic transformations, and to implement a computerized tool that will automate the security analysis of transformations.

Acknowledgments

The research in this report has been funded by the Research Council of Norway through the project SECURIS (152839/220). We would like to thank Fabio Massacci and Jan Øyvind Aagedal for useful comments and suggestions.

References


REFERENCES


REFERENCES


A Glossary of Symbols

A.1 Set notation

<table>
<thead>
<tr>
<th>Set*</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P(A) )</td>
<td>( { X \mid X \subseteq A } ) The power set of ( A ).</td>
</tr>
<tr>
<td>( A^\omega )</td>
<td>( { } ) The set of all finite and infinite sequences over the set ( A ).</td>
</tr>
<tr>
<td>( l \in L )</td>
<td>( { } ) The set of all lifelines.</td>
</tr>
<tr>
<td>( si \in SI )</td>
<td>( { } ) The set of all signals.</td>
</tr>
<tr>
<td>( m \in M )</td>
<td>( { !, ? } ) The set of all messages.</td>
</tr>
<tr>
<td>( K )</td>
<td>( { !, ? } ) The set of all kinds.</td>
</tr>
<tr>
<td>( e \in E )</td>
<td>( { !, ? } ) The set of all events.</td>
</tr>
<tr>
<td>( h \in H )</td>
<td>( \mathbb{E} ) The trace universe.</td>
</tr>
<tr>
<td>( p, n, b \subseteq H )</td>
<td>( { !, ? } ) A set of positive, negative, and behavior traces, respectively.</td>
</tr>
<tr>
<td>( H_a \subseteq H )</td>
<td>( { !, ? } ) The abstract trace universe.</td>
</tr>
<tr>
<td>( H_c \subseteq H )</td>
<td>( { !, ? } ) The concrete trace universe.</td>
</tr>
<tr>
<td>( f \in F )</td>
<td>( { !, ? } ) The set of functions that conform to (2) and (1) Sect. 5.1.</td>
</tr>
<tr>
<td>( a, c \in O )</td>
<td>( \mathbb{P}(H) \times \mathbb{P}(H) ) The set of all interaction obligations.</td>
</tr>
<tr>
<td>( a, d, s \in S )</td>
<td>( \mathbb{P}(L) ) The set of all systems.</td>
</tr>
</tbody>
</table>

*By the notation \( a \in A \) we understand that \( A \) is ranged over by \( a \).*

A.2 Traces

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle \rangle )</td>
<td>The empty trace</td>
<td>( \langle \rangle )</td>
</tr>
<tr>
<td>( \langle !a, ?a \rangle )</td>
<td>The trace with two events</td>
<td>( \langle !a, ?a \rangle )</td>
</tr>
<tr>
<td>#h</td>
<td>The length of trace ( h )</td>
<td>#( \langle !a, ?a \rangle ) = 2</td>
</tr>
<tr>
<td>( A \otimes h )</td>
<td>The trace ( h ) restricted to set ( A )</td>
<td>( { !a, !a } \otimes \langle !a, !b, ?a, ?b \rangle = \langle !a, !a \rangle )</td>
</tr>
<tr>
<td>( h_1 \sqsubseteq h_2 )</td>
<td>( h_1 ) is prefix or equal to ( h_2 )</td>
<td>( \langle !a, !b \rangle \sqsubseteq \langle !a, !b, ?a \rangle ) yields true</td>
</tr>
<tr>
<td>( h[n] )</td>
<td>The event in trace ( h ) at index ( n )</td>
<td>( \langle !a, ?a, !b, ?b \rangle [2] = ?a )</td>
</tr>
<tr>
<td>( h_1 \sqcap h_2 )</td>
<td>Concatination of ( h_1 ) and ( h_2 )</td>
<td>( \langle !a, ?a \rangle \sqcap \langle !b, ?b \rangle ) yields ( \langle !a, ?a, !b, ?b \rangle )</td>
</tr>
</tbody>
</table>
A.3 Auxiliar Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Def.</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k_e \in E \rightarrow K$</td>
<td>$k_e \equiv {! } \times M$ if $e \in {! } \times M$; otherwise.</td>
<td>The kind (output or input) of event $e$.</td>
</tr>
<tr>
<td>$tr_e \in E \rightarrow L$</td>
<td>$tr_e \equiv t$, where $e \in {t} \times L \times SI$</td>
<td>The transmitter of event $e$.</td>
</tr>
<tr>
<td>$re_e \in E \rightarrow L$</td>
<td>$re_e \equiv r$, where $e \in L \times {r} \times SI$</td>
<td>The transmitter of event $e$.</td>
</tr>
<tr>
<td>$e_s \in S \rightarrow P(E)$</td>
<td>$e_s \equiv { e \in E \mid (k_e =! \land tr_e \in s) \lor (k_e =? \land re_e \in s) }$</td>
<td>The set of events that can occur on the lifelines of system $s$.</td>
</tr>
<tr>
<td>$rt_s \in S \rightarrow P(E)$</td>
<td>$rt_s \equiv { e \in E \mid tr_e \in s \lor re_e \in s }$</td>
<td>The set of events whose message can be sent to or from the lifelines in $s$.</td>
</tr>
<tr>
<td>$e_{l_1, l_2} \in L \times L \rightarrow P(E)$</td>
<td>$e_{l_1, l_2} \equiv E_{l_1} \cap RT{l_2}$</td>
<td>The set of events on lifeline $l_1$ whose message can be sent from $l_1$ to $l_2$ or from $l_2$ to $l_1$.</td>
</tr>
</tbody>
</table>

B Proofs

The proofs in this section are structured in a way similar to that of [23], where hierarchical structure is used to demonstrate how each proof step is justified.

B.1 Theorem 1: Information flow property preserving behavior refinement

Theorem 1: The ignorance property is preserved by refinement.

Assume: 1. $o^{(d_a, s)} o'$
2. $d_a \rightarrow d_c$

Prove: $o^{(d_a, s)} o'$

\[\langle 1 \rangle\] Assume: 1.1 $h_0 \in o \triangleright (d_a, s)$
1.2 $\beta_1 \in \nabla (d_a, s)$
1.3 $\neg (o \triangleright h_0 o')$

Prove: $\exists \beta_2 \in \nabla (d_c, s) : o \triangleright_2 h_0 o' \land \beta_2 \equiv o \triangleright h_0 \beta_2$

\[\langle 2 \rangle\] Choose $a_1 \in [d_a]$ and $c_1 \in [d_c]$ such that $a_1 \rightarrow c_1$ and $\beta_1 = \nabla (c_1, s)$

Proof: By assumption 1.2 and definition of $\rightarrow$ (Def. 16) and $\nabla$ (Def. 9).

\[\langle 3 \rangle\] Choose $a_1 \in \nabla (d_a, s)$ such that $o_1 = \nabla (a_1, s)$

Proof: By $\langle 3 \rangle$ 1 and definition of $\nabla$ (Def. 9).

\[\langle 4 \rangle\] $\neg (o \triangleright a_1 h_o o')$
Choose \( d \) such that \( \text{rt}\, o \otimes h = h_o \) and \( \text{rt}\, o' \otimes h \neq \emptyset \)

**Proof:** By assumption 1.3 and definition of \( \text{rt} \) (Def. 13).

\( 4.2 \). \( h \in \alpha_1 \)

**Proof:** By (3)1, (3)2, (4)1, and definition of \( \leftarrow \) (Def. 15).

(3)3. Q.E.D.

**Proof:** By (4)1, (4)2, and definition of \( \text{rt} \) (Def. 13).

(3)4. Q.E.D.

**Proof:** By (3)3, assumption 1 and definition of \( \text{rt} \) (Def. 14).

(2)2. Choose \( a_2 \in \{ d_a \} \) and \( c_2 \in \{ d_c \} \) such that \( a_2 \rightarrow c_2 \) and \( \omega_2 = \nabla(a_2, s) \)

**Proof:** By (2)1 and assumption 2, and definition of \( \nabla \) (Def. 9).

(2)3. Choose \( \beta_2' \in \nabla(d_c, s) \) such that \( \beta_2' = \nabla(c_2, s) \)

**Proof:** By (2)2 and definition of \( \nabla \) (Def. 9).

(2)4. \( o \downarrow \langle \beta_2' \rangle \)

**Assume:** (a) \( \neg (o \downarrow \langle \beta_2' \rangle) \)

**Prove:** False

(3)1. \( \exists h' \in o \downarrow \alpha_2 : rt\, o' \otimes h' \neq \emptyset \)

(4)1. \( \exists h' : rt\, o' \otimes h' \neq \emptyset \)

(5)1. \( h' \in o \downarrow \alpha_2 \)

**Proof:** By assumption (a) and definition of \( \text{rt} \) (Def. 13).

(5)2. \( h \in o \downarrow \alpha_2 \)

**Proof:** By (2)2, (2)3, (5)1 and definition of \( \leftarrow \) (Def. 15).

(5)3. Q.E.D.

**Proof:** Select \( h' \) such that \( h' = h \), then (4)1 holds by (5)1 and (5)2.

(4)2. Q.E.D.

**Proof:** By (4)1 and definition of \( \text{rt} \) (Def. 13).

(3)2. Q.E.D.

**Proof:** \( (3)1 \) contradicts (2)1, hence assumption (a) is false and (2)4 holds.

(2)5. \( \beta_2' = o \downarrow \alpha_2 \beta_2' \)

**Proof:** Select \( \beta_2' \) such that \( \beta_2 = \beta_2' \), then (1)1 holds by (2)4 and (2)5.

(1)2. Q.E.D.

**Proof:** By (1)1 and definition of \( \text{rt} \) (Def. 14).

### B.2 Lemma 1: Transformation is transitive

**Lemma 1:** \( \leftarrow \) is transitive

**Assume:** 1. \( d \leftarrow T_1 \ d T_1 \)

2. \( d T_1 \leftarrow T_2 \ d T_2 \)

**Prove:** \( d \leftarrow T_2 o T_1 \ d T_2 \)

(1)1. \( \left[ d T_2 \right] = \{ o T_2 \in \Omega \mid o \leftarrow f \text{ } o T_2 \wedge o \in \left[ d \right] \wedge f \in \left[ T_2 \circ T_1 \right] \} \)

(2)1. \( \exists (p T_1, n T_1) \in \left[ d T_1 \right] : (p, n) \leftarrow f (p T_2, n T_2) \) for arbitrary \( (p, n) \in \left[ d \right] \)

**Proof:** By definition of \( o \).

(3)1. \( \exists f_1 \in \left[ T_1 \right] \) and \( f_2 \in \left[ T_2 \right] \) such that \( f = f_2 \circ f_1 \)

**Proof:** By assumption 1 and definition of \( \leftarrow \) (Def. 17).
3. Choose \((p_{T_2}, n_{T_2})\) such that \((p_{T_1}, n_{T_1}) \leadsto f_1 (p_{T_2}, n_{T_2})\)
   \(\text{Proof: By assumption 2 and definition of } \leadsto \text{ (Def. 17)}.\)

3.4. \((p, n) \leadsto f \circ f_1 (p_{T_2}, n_{T_2})\)
   \(\text{Proof: By (3)2, (3)3, and Lemma 1.1 (Sect. B.2.1)}.\)

3.5. Q.E.D.
   \(\text{Proof: By (3)1 and (3)4}.\)

2. \(\exists (p, n) \in \{d\} : \exists f \in \{T_2 \circ T_1\} : (p, n) \leadsto f (p_{T_2}, n_{T_2})\) for arbitrary
   \((p_{T_2}, n_{T_2})\) in \([d_{T_2}]\)

3. Choose \(f_2 \in \{T_2\}\) and \((p_{T_1}, n_{T_1}) \in \{d_{T_1}\}\) such that \((p_{T_1}, n_{T_1}) \leadsto f_2 (p_{T_2}, n_{T_2})\)
   \(\text{Proof: By assumption 2 and definition of } \leadsto \text{ (Def. 17)}.\)

3.2. Choose \(f_1 \in \{T_1\}\) and \((p, n) \in \{d\}\) such that \((p, n) \leadsto f_1 (p_{T_1}, n_{T_1})\)
   \(\text{Proof: By assumption 1 and definition of } \leadsto \text{ (Def. 17)}.\)

3.3. \((p, n) \leadsto f \circ f_1 (p_{T_2}, n_{T_2})\)
   \(\text{Proof: By (3)1, (3)2, and Lemma 1.1 (Sect. B.2.1)}.\)

3.4. \(f_2 \circ f_1 \in \{T_2 \circ T_1\}\)
   \(\text{Proof: By (3)1, (3)2, and definition of } \circ.\)

3.5. Q.E.D.
   \(\text{Proof: By (3)3 and (3)4}.\)

2.3. Q.E.D.
   \(\text{Proof: By (2)1 and (2)2}.\)

(1)2. Q.E.D.
   \(\text{Proof: By definition of } \leadsto \text{ (Def. 17)}.\)

### B.2.1 Lemma 1.1

**Assume:**
1. \((p, n) \leadsto f_1 (p_{T_1}, n_{T_1})\)
2. \((p_{T_1}, n_{T_1}) \leadsto f_2 (p_{T_2}, n_{T_2})\)

**Prove:** \((p, n) \leadsto f_2 \circ f_1 (p_{T_2}, n_{T_2})\)

**Let:** \(f \triangleq f_2 \circ f_1\)

1. \(\mathcal{H}_{T_2} \setminus n_{T_2} = \{f(h) | h \in \mathcal{H} \setminus n\} \setminus p_{T_2} = \{f(h) | h \in p\}\)
2. \(\forall h \in \mathcal{H} \setminus n : \exists h_{T_1} \in \mathcal{H}_{T_1} \setminus n_{T_1} : h_{T_1} = f(h)\)
3. \(\exists h_{T_2} \in \mathcal{H}_{T_2} \setminus n_{T_2} : h_{T_2} = f(h)\) for arbitrary \(h \in \mathcal{H} \setminus n\)
4.1. Choose \(h_{T_2} \in \mathcal{H}_{T_2} \setminus n_{T_2}\) such that \(h_{T_2} = f_1(h)\)
   \(\text{Proof: By assumption 1 and definition of } \leadsto \text{ (Def. 17)}.\)

4.2. Choose \(h_{T_2} \in \mathcal{H}_{T_2} \setminus n_{T_2}\) such that \(h_{T_2} = f_2(h_{T_1})\)
   \(\text{Proof: By assumption 2 and definition of } \leadsto \text{ (Def. 17)}.\)

4.3. Q.E.D.
   \(\text{Proof: By (4)1 and (4)2 and definition of } f.\)

3.2. Q.E.D.
   \(\text{Proof: By (3)1}.\)

2. \(\forall h_{T_2} \in \mathcal{H}_{T_2} \setminus n_{T_2} : \exists h \in \mathcal{H} \setminus n : h_{T_2} = f(h)\)
3.1. \(\exists h \in \mathcal{H} \setminus n : h_{T_2} = f(h)\) for arbitrary \(h_{T_2} \in \mathcal{H}_{T_2} \setminus n_{T_2}\)
4.1. Choose \(h_{T_1} \in \mathcal{H}_{T_1} \setminus n_{T_1}\) such that \(h_{T_1} = f_2(h_{T_1})\)
   \(\text{Proof: By assumption 2 and definition of } \leadsto \text{ (Def. 17)}.\)

4.2. Choose \(h \in \mathcal{H} \setminus n\) such that \(h_{T_1} = f_1(h)\)
   \(\text{Proof: By assumption 1 and definition of } \leadsto \text{ (Def. 17)}.\)

4.3. Q.E.D.
   \(\text{Proof: By (4)1 and (4)2 and definition of } f.\)
\[ \langle 3 \rangle 2. \text{Q.E.D.} \]
Proof: By \langle 3 \rangle 1.
\[ \langle 2 \rangle 3. \forall h \in p : \exists h_{T_2} \in p_{T_2} : h_{T_2} = f(h) \]
\[ \langle 3 \rangle 1. \exists h_{T_2} \in p_{T_2} : h_{T_2} = f(h) \text{ for arbitrary } h \in p \]
\[ \langle 4 \rangle 1. \text{Choose } h_{T_1} \in p_{T_1} \text{ such that } h_{T_1} = f_1(h) \]
Proof: By assumption 1 and definition of \( \Rightarrow \) (Def. 17).
\[ \langle 4 \rangle 2. \text{Choose } h_{T_2} \in p_{T_2} \text{ such that } h_{T_2} = f_2(h_{T_1}) \]
Proof: By assumption 2 and definition of \( \Rightarrow \) (Def. 17).
\[ \langle 4 \rangle 3. \text{Q.E.D.} \]
Proof: By \langle 4 \rangle 1 and \langle 4 \rangle 2 and definition of \( f \).
\[ \langle 3 \rangle 2. \text{Q.E.D.} \]
Proof: By \langle 3 \rangle 1.
\[ \langle 2 \rangle 4. \forall h_{T_2} \in p_{T_2} : \exists h \in p : h_{T_2} = f(h) \]
\[ \langle 3 \rangle 1. \exists h \in p : h_{T_2} = f(h) \text{ for arbitrary } h_{T_2} \in p_{T_2} \]
\[ \langle 4 \rangle 1. \text{Choose } h_{T_1} \in p_{T_1} \text{ such that } h_{T_1} = f_2(h_{T_1}) \]
Proof: By assumption 2 and definition of \( \Rightarrow \) (Def. 17).
\[ \langle 4 \rangle 2. \text{Choose } h \in p \text{ such that } h_{T_1} = f_1(h) \]
Proof: By assumption 1 and definition of \( \Rightarrow \) (Def. 17).
\[ \langle 4 \rangle 3. \text{Q.E.D.} \]
Proof: By \langle 4 \rangle 1 and \langle 4 \rangle 2 and definition of \( f \).
\[ \langle 3 \rangle 2. \text{Q.E.D.} \]
Proof: By \langle 3 \rangle 1.
\[ \langle 2 \rangle 5. \text{Q.E.D.} \]
Proof: By \langle 2 \rangle 1 - \langle 2 \rangle 4.
\[ \langle 1 \rangle 2. \text{Q.E.D.} \]
Proof: By definition of \( \Rightarrow \) (Def. 17).

\section*{B.3 Lemma 2: Refinement is transitive modulo translation}

\textbf{Lemma 2:} \( \sim \) is transitive modulo translation.
\textbf{Assume:} 1. \( d \sim_{T_1} d'_{T_1} \)
\[ 2. d''_{T_1} \sim_{T_2} d''_{T_2} \]
\textbf{Prove:} \( d \sim_{T_2 \circ T_1} d''_{T_2} \)
\[ \langle 1 \rangle 1. \exists d_{T_2} \in D : d \sim_{T_2 \circ T_1} d_{T_2} \wedge d_{T_2} \sim d''_{T_2} \]
\[ \langle 2 \rangle 1. \text{Choose } d_{T_1} \in D \text{ such that} \]
\[ \text{(a) } d \sim_{T_1} d_{T_1} \text{ and} \]
\[ \text{(b) } d_{T_1} \sim d''_{T_1} \]
Proof: By assumption 1 and definition of \( \sim \) (Def. 20).
\[ \langle 2 \rangle 2. \text{Choose } d''_{T_2} \in D \text{ such that} \]
\[ \text{(a) } d''_{T_1} \sim_{T_2} d''_{T_2} \text{ and} \]
\[ \text{(b) } d''_{T_2} \sim d''_{T_2} \]
Proof: By assumption 2 and definition of \( \sim \) (Def. 20).
\[ \langle 2 \rangle 3. \text{Choose } d_{T_2} \in D \text{ such that} \]
\[ \text{(a) } d_{T_1} \sim_{T_1} d_{T_2} \text{ and} \]
\[ \text{(b) } d_{T_2} \sim d_{T_2} \]
Proof: By \langle 2 \rangle 1 \( \text{(b) } \), \langle 2 \rangle 2 \( \text{(a) } \), and Lemma 2.1 (Sect. B.3.1).
\[ \langle 2 \rangle 4. d \sim_{T_2 \circ T_1} d_{T_2} \]
Proof: By \langle 2 \rangle 1 \( \text{(a) } \), \langle 2 \rangle 3 \( \text{(a) } \), and Lemma 1 (Sect. B.2).
(2.5) \( d_{T_2} \rightsquigarrow d'_{T_2} \)

**Proof:** By (2.3) (b), (2.2) (b), and transitivity of \( \rightsquigarrow \) (proved in [14]).

(2.6) Q.E.D.

**Proof:** By (2.4) and (2.5).

(1.2) Q.E.D.

**Proof:** By definition of \( \rightsquigarrow \) (Def. 20).

### B.3.1 Lemma 2.1

**Assume:**

1. \( d \rightsquigarrow d' \)
2. \( d' \rightsquigarrow_T d''_T \)

**Prove:** \( d \rightsquigarrow_T d''_T \)

(1.1) \( \{ d''_T \} = \{ o'_{d_2} \in \mathcal{O} \mid o \rightsquigarrow f o'_{d_2} \land o \in \{ d \} \land f \in \{ T \} \} \)

(2.1) \( \forall f \in \{ T \} : \forall o \in \{ d \} : \exists o' \in \{ d''_T \} : o \rightsquigarrow f o' \)

(3.1) \( \exists o' \in \{ d''_T \} : o \rightsquigarrow f o' \) for arbitrary \( f \in \{ T \} \) and \( o \in \{ d \} \)

(4.1) Choose \( o' \in \{ d'' \} \) such that \( o \rightsquigarrow o' \)

**Proof:** By assumption 1 and definition of \( \rightsquigarrow \) (Def. 15).

(4.2) Choose \( o' \in \{ d''_T \} \) such that \( o' \rightsquigarrow_T o' \)

**Proof:** By assumption 2 and definition of \( \rightsquigarrow_T \) (Def. 17).

(4.3) \( o \rightsquigarrow_T o' \)

**Proof:** By (4.1), (4.2), and Lemma 2.2 (Sect. B.3.2).

(4.4) Q.E.D.

**Proof:** By (4.3).

(3.2) Q.E.D.

**Proof:** By \( \forall \)-rule.

(2.2) \( \forall o' \in \{ d''_T \} : \exists f \in \{ T \} : \exists o \in \{ d \} : o \rightsquigarrow f o' \)

(3.1) \( \exists f \in \{ T \} : \exists o \in \{ d \} : o \rightsquigarrow f o' \) for arbitrary \( o' \in \{ d''_T \} \)

(4.1) Choose \( o' \in \{ d'' \} \) and \( f \in \{ T \} \) such that \( o' \rightsquigarrow_T o' \)

**Proof:** By assumption 2 and definition of \( \rightsquigarrow_T \) (Def. 17).

(4.2) Choose \( o \in \{ d \} \) such that \( o \rightsquigarrow o' \)

**Proof:** By assumption 1 and definition of \( \rightsquigarrow \) (Def. 15).

(4.3) \( o \rightsquigarrow_T o' \)

**Proof:** By (4.2), (4.1), and Lemma 2.2 (Sect. B.3.2).

(4.4) Q.E.D.

**Proof:** By (4.3).

(3.2) Q.E.D.

**Proof:** By \( \forall \)-rule.

(2.3) Q.E.D.

**Proof:** Trivial.

(1.2) Q.E.D.

**Proof:** By (1.1) and definition of \( \rightsquigarrow \) (Def. 20).

### B.3.2 Lemma 2.2

**Assume:**

1. \( (p_a, n_a) \rightsquigarrow (p'_a, n'_a) \)
2. \( (p'_a, n'_a) \rightsquigarrow_T (p'_c, n'_c) \)

**Prove:** \( (p_a, n_a) \rightsquigarrow_T (p'_c, n'_c) \)

(1.1) \( \exists (p_c, n_c) \in \mathcal{O} : (p_a, n_a) \rightsquigarrow_T (p_c, n_c) \land (p_c, n_c) \rightsquigarrow (p'_c, n'_c) \)
(2) 1. \( \exists (p_e, n_c) \in o : \)
\( H_e \setminus n_c = \{ f(h) \mid h \in (H_a \setminus n_a) \} \land \)
\( p_e = \{ f(h) \mid h \in p_a \} \land \)
\( n_c \subseteq n'_c \land p_e \subseteq p'_e \cup n'_c \)

(3) 1. Choose \( (p_e, n_c) \in o \) such that \( H_e \setminus n_c = \{ f(h) \mid h \in (H_a \setminus n_a) \} \) and \( p_e = \{ f(h) \mid h \in p_a \} \)

Proof: By definition of \( o \).

(3) 2. \( n_c \subseteq n'_c \)

   (4) 1. \( H_a \setminus n'_a \subseteq H_a \setminus n_a \)

   Proof: \( n_a \subseteq n'_a \) by assumption 1 and definition of \( \rightsquigarrow \) (Def 15), hence \( n_c \subseteq n'_c \).

   (4) 2. \( \{ f(h') \mid h' \in (H_a \setminus n'_a) \} \subseteq \{ f(h) \mid h \in (H_a \setminus n_a) \} \)

   Proof: By (4) 1 since \( f \) is a function and hence deterministic.

   (4) 3. \( H_e \setminus n'_c \subseteq H_e \setminus n_c \)

   Proof: By (4) 2, assumption 2, (3) 1, and definition of \( \iff \) (Def. 17).

(4) 4. Q.E.D.

Proof: By (4) 3 and definition of \( H_e \).

(3) 3. \( p_e \subseteq p'_e \cup n'_c \)

Assume: 1.1. \( p_c \not\subseteq p'_c \cup n'_c \)

Proof: False

   (5) 1. \( \exists h \in p_a : h \not\in (p'_c \cup n'_c) \)

   Proof: By assumption 1.1.

   (5) 2. Choose \( h_e \in p_e \) such that \( h_e \not\in (p'_c \cup n'_c) \)

   Proof: By (5) 1 and (3) 1.

   (5) 3. \( h_a \not\in p'_a \)

   Proof: By assumption 2, definition of \( \rightsquigarrow \) (Def. 17), and modus tollens.

   (5) 4. \( h_a \not\in n'_c \)

   Proof: By assumption 2, definition of \( \rightsquigarrow \) (Def. 17), and modus tollens.

   (5) 5. Q.E.D.

   Proof: By (5) 2, (5) 3, and (5) 4.

(4) 2. Q.E.D.

Proof: (4) 1 contradicts assumption 1 by definition of \( \rightsquigarrow \) (Def. 15).

Thus assumption 1.1. is false, and (3) 3 must hold.

(3) 4. Q.E.D.

Proof: By (3) 1, (3) 2, and (3) 3.

(2) 2. Q.E.D.

Proof: By (2) 1, definition of \( \iff \) (Def. 17), and definition of \( \rightsquigarrow \) (Def. 15).

(1) 2. Q.E.D.

Proof: By definition of \( \rightsquigarrow \) (Def. 19).

B.4 Theorem 2: Ignorance preserving transformation

Theorem 2: The ignorance property is preserved by transformation.

Assume: 1. \( a^{(d_a, x)} a' \)

2. \( d_a \rightsquigarrow_r d_e \)
PROVE: $o \downarrow (d_c, s) o'$

(1.1) Assume: 1.1 $h_c \in o \triangleright (d_c, s)$
1.2 $c_1 \in \nabla (d_c, s)$
1.3 $\neg (a \triangleleft_{c_1} h_c \ o')$

PROVE: $\exists c_2 \in \nabla (d_c, s) : o \triangleleft_{c_2} h_c \ o' \land c_2 = o \triangleleft h_c \ c_2$

Let: $\omega \triangleleft (\text{rt} . o \cap \text{e} . s)$, and $\omega' \triangleleft (\text{rt} . o' \cap \text{e} . s)$

(2.1) Choose $(p^1_c, n^1_c) \in [d_c]$ such that $c_1 = \nabla ((p^1_c, n^1_c), s)$

PROOF: By assumption 1.2 and definition of $\nabla$ (Def. 9).

(2.2) Choose $(p^1_c, n^1_c) \in [d_a]$ and $f \in [T]$ such that $(p^1_c, n^1_c) \hookleftarrow f (p^1_c, n^1_c)$

PROOF: By (2.1), assumption 2, and definition of $\hookleftarrow$ (Def 18).

(2.3) Choose $\beta_1 \in \mathcal{H}_c \setminus n^1_c$ such that $h_c = \omega \odot \beta_1$ and $\omega' \odot \beta_1 \neq \{\}$

(3.1) Choose $h_c^k \in o \triangleleft_{h_c} c_1 : \text{rt} . o' \odot h_c^k \neq \{\}$

PROOF: By assumption 1.3 and definition of $\triangleleft$ (Def. 13).

(3.2) Choose $\beta \in \mathcal{H}_c \setminus n^1_c$ such that $e . s \odot \beta = h_c^k$

PROOF: By assumption 1.2, (2.1) and definition of $\triangleleft$ (Def. 12).

(3.3) $(e . s \cap \text{rt} . o) \odot \beta = h_c$

(4.1) $\text{rt} . o \odot h_c^k = h_c$

PROOF: By (3.1) and definition of $\triangleleft$ (Def 12).

(4.2) Q.E.D.

PROOF: It is easy to see that $A \odot (B \odot h) = (A \cap B) \odot h$ for arbitrary sets $A$ and $B$ and trace $h$. Thus (3.3) holds by (3.2) and (4.1).

(3.4) $(e . s \cap \text{rt} . o') \odot \beta \neq \{\}$

PROOF: By (3.1) and (3.3).

(3.5) Q.E.D.

PROOF: By (3.2), (3.3), (3.4) and definition of $\omega$ and $\omega'$.

(4.1) $\omega' \odot \beta_1 = \{\}$

(4.2) Q.E.D.

PROOF: By (4.1), assumption (a), and Lemma 2.4 (Sect. B.4.3).

(3.2) Q.E.D.

PROOF: Assumption (a) cannot hold since (3.1) contradicts (2.3), hence (2.5) must hold.

Let: $h_a \triangleleft \omega \odot \alpha_1$

(2.6) $h_c = f(h_a)$

PROOF: By (2.3), (2.4), definition of $h_a$ and Lemma 2.1 (Sect. B.4.1).

(2.7) $h_a \in o \triangleright (d_a, s)$

PROOF: By definition of $\triangleright$ (Def. 11) and $\omega$.

(2.8) Choose $a_2 \in \nabla (d_a, s)$ such that $o \triangleleft_{a_2} h_a \ o' \land a_2 = o \triangleleft h_a \ a_2$

(3.1) Choose $\alpha_1 \in \nabla (d_a, s)$ such that $a_1 = \nabla ((p^1_a, n^1_a), s)$

PROOF: By (2.2) and definition of $\nabla$ (Def. 9).

(3.2) $\neg (o \triangleleft_{a_1} h_a \ o')$

(4.1) $\exists h \in o \triangleleft h_a : \text{rt} . o' \odot h \neq \{\}$

(5.1) $(e . s \odot \alpha_1) \in o \triangleleft h_a \ a_1$
(6.1) $(e.s \circ \alpha_1) \in a_1$
PROOF: By (2.4), (3.1) and definition of $\nabla$ (Def. 9).

(6.2) $rt.o \circ (e.s \circ \alpha_1) = h_a$
PROOF: By definition of $h_a$. 

(6.3) Q.E.D.
PROOF: By (6.1) and (6.2) and definition of $\lhd$ (Def. 12).

(5.2) $rt.o' \circ (e.s \circ \alpha_1) \neq \emptyset$
PROOF: It is easy to see that $A \circ (B \circ h) = (A \cap B) \circ h$ for arbitrary sets of events $A$ and $B$ and trace $h$. Hence (5.2) holds by (2.5) and definition of $\omega$.

(5.3) Q.E.D.
PROOF: Select $h$ such that $h = e.s \circ \alpha_1$, then (4.1) holds by (5.1) and (5.2).

(4.2) Q.E.D.
PROOF: By (4.1) and definition of $\lhd$ (Def. 13).

(3.3) Q.E.D.
PROOF: By (3.2), (2.7), assumption 1, and definition of $\lhd$ (Def. 14).

(2.9) Choose $(p_2^c, n_2^c) \in [d_a]$ such that $a_2 = \nabla((p_2^c, n_2^c), s)$
PROOF: By (2.8) and definition of $\nabla$ (Def. 9).

(2.10) Choose $(p_2^c, n_2^c) \in [d_a]$ such that $(p_2^c, n_2^c) \mapsto f(p_2^c, n_2^c)$
PROOF: By (2.9), assumption 2, and definition of $\mapsto$ (Def. 18).

(2.11) Choose $c_2 \in \nabla(d_a, s)$ such that $c_2 = \nabla((p_2^c, n_2^c), s)$
By (2.10) and definition of $\nabla$ (Def. 9).

(2.12) $c_2 = o <_{h_e} c_2$
ASSUME: $(a) \neg(c_2 = o <_{h_e} c_2)$
PROOF: False.

(3.1) $a_2 \neq o <_{h_e} a_2$

(4.1) $\exists h \in a_2 : (rt.o \circ h) \neq h_a$

(5.1) Choose $h_b^b \in c_2$ such that $h_b \neq (rt.o \circ h_b^b)$
PROOF: By assumption (a) and definition of $\lhd$ (Def. 12).

(5.2) Choose $\beta \in H_e \setminus n_2^c$ such that $e.s \circ \beta = h_b^b$
PROOF: By (2.11), (5.1), and definition of $\nabla$ (Def. 8).

(5.3) $\omega \circ \beta \neq h_e$
PROOF: By (5.1), (5.2), and definition of $\omega$ and $\circ$.

(5.4) Choose $\alpha \in H_a \setminus n_2^c$ such that $\beta = f(\alpha)$
PROOF: By (2.10), (5.2), and definition of $\mapsto$ (Def. 17).

(5.5) $\omega \circ \alpha \neq h_a$

(6.1) $f(\omega \circ \alpha) \neq f(h_a)$
PROOF: 
\begin{align*}
\omega \circ \beta & \neq h_e \quad \text{[By (5.3)]} \\
 f(\omega \circ \alpha) & \neq h_e \quad \text{[By (5.4) and Lemma 2.1]} \\
 f(\omega \circ \alpha) & \neq f(h_a) \quad \text{[By (2.6)]}
\end{align*}

(6.2) Q.E.D.

$f(h_1) \neq f(h_2) \Rightarrow h_1 \neq h_2$ because $f$ is a function.

(5.6) $(e.s \circ \alpha) \in a_2$
PROOF: By (2.9), (5.4), and definition of $\nabla$ (Def. 8).

(5.7) $rt.o \circ (e.s \circ \alpha) \neq h_a$
PROOF: By (5.5) and definition of $\omega$ and $\circ$.

(5.8) Q.E.D.
Proof: Select $h$ such that $h = (e.s \circ \alpha)$, then (4)1 holds by (5)6 and (5)7.

(4)2. Q.E.D.

Proof: By (4)1 and definition of $\prec$ (Def. 12).

(3)2. Q.E.D.

Proof: Assumption (a) cannot hold since (3)1 contradicts (2)8. Hence (2)12 holds.

(2)13. $\forall h \in o \triangleleft h \in o \cap \beta$ : \(\text{rt}.\beta \circ h = \emptyset\)

(3)1. $\forall h \in o \triangleleft h \in o \cap \beta$ : \(\text{rt}.\beta \circ h = \emptyset\)

(4)1. $\text{rt}.\beta \circ h = \emptyset$ for arbitrary $h \in o \triangleleft h \in o \cap \beta$

(5)1. Choose $\beta_2 \in \mathcal{H}_e \setminus n^1_s$ such that $e.s \circ \beta_2 = h$

Proof: By (2)11 and definition of $\prec$ (Def. 12).

(5)2. Choose $\alpha_2 \in \mathcal{H}_e \setminus n^1_s$ such that $\beta_2 = f(\alpha_2)$

Proof: By (2)10, (5)1, and definition of $\Longleftrightarrow$ (Def. 17).

(5)3. $\omega' \circ \beta_2 = \emptyset$

(6)1. $\omega' \circ \alpha_2 = \emptyset$

Proof: By (2)11, (2)9, (5)2, and definition of $\Longleftrightarrow$ (Def. 13) and $\omega'$.

(6)2. Q.E.D.

Proof: By (5)2, (6)1 and Lemma 2.3 (Sect. B.4.3).

(5)4. $\text{rt}.\beta \circ (e.s \circ \beta_2) = \emptyset$

Proof: By (5)3 and definition of $\omega'$ and $\circ$.

(5)5. Q.E.D.

Proof: By (5)1, and (5)4.

(4)2. Q.E.D.

Proof: By (4)1 and $\forall$-rule.

(3)2. Q.E.D.

Proof: By (3)1 and definition of $\Longleftrightarrow$ (Def. 13).

(2)14. Q.E.D.

Proof: By (2)11, (2)12, and (2)13.

(1)2. Q.E.D.

Proof: By (1)1 and definition of $\Longleftrightarrow$ (Def. 14).

B.4.1 Lemma 2.1

Assume: 1. $\beta = f(\alpha)$ for $f \in \mathcal{F}$

Prove: $\omega \circ \beta = f(\omega \circ \alpha)$ where $\omega \triangleleft (\text{rt}.\circ \cap e.s)$ for arbitrary $s, o \in S$

Proof sketch: By induction over the length of $\alpha$ we show that (1) $\omega \circ f(\alpha) = f(\omega \circ \alpha)$ holds for the base case where $\# \alpha = 0$ and that (1) holds for $\# \alpha = n$ (induction hypothesis) implies that (1) holds for $\# \alpha = n + 1$.

(1)1. $\omega \circ f(\alpha) = f(\omega \circ \alpha)$

(2)1. Case: $\# \alpha = 0$, i.e. $\alpha = \emptyset$ (base case).

Proof:

\[
\begin{align*}
\omega \circ f(\alpha) &= f(\omega \circ \alpha) \\
\omega \circ f(\emptyset) &= f(\omega \circ \emptyset) & [\alpha = \emptyset] \\
\omega \circ f(\{\}) &= f(\{\}) & [\text{by definition of } \circ] \\
\omega \circ \{\} &= \{\} & [\text{by Lemma 2.3 (Sect. B.4.3).}] \\
\{\} &= \{\} & [\text{by definition of } \circ . ]
\end{align*}
\]

(2)2. Case: $\# \alpha = n + 1$

Assume: $\omega \circ f(\alpha_p) = f(\omega \circ \alpha_p)$ where $\alpha_p \prec e = \alpha$ such that $e = \alpha[n + 1]$ (induction hypothesis).
\textbf{B. PROOFS}

\begin{align*}
\text{Proof:} & \quad \omega \odot f(\langle e \rangle) = f(\omega \odot \langle e \rangle) \quad \text{[Lemma 2.2]} \\
\omega \odot f(\alpha_p) & = f(\omega \odot \alpha_p) \quad \text{[Induct. hyp.]} \\
\omega \odot f(\alpha_p) & = f((\omega \odot \alpha_p) \odot (\omega \odot \langle e \rangle)) \quad \text{[*]} \\
\omega \odot f(\alpha_p) & = f(\omega \odot \{\alpha_p \odot \langle e \rangle\}) \quad \text{[Def. of } \odot \text{]} \\
\omega \odot f(\alpha) & = f(\omega \odot \alpha) \quad \text{[Induct. hyp.]} \\
\end{align*}

* These steps are valid because \( f \in \mathcal{F} \), thus \( f \) fulfills requirement (2) (see Sect. 5.1) by definition of \( \mathcal{F} \).

\( \langle 2 \rangle \) 3. Q.E.D.

\textbf{Proof:} By \( \langle 2 \rangle \) 1 and \( \langle 2 \rangle \) 2.

\textbf{B.4.2 Lemma 2.2}

\textbf{Proof:} \( \omega \odot f(\langle e \rangle) = f(\omega \odot \langle e \rangle) \) where \( \omega \triangleq (\text{RT}. o \cap \text{E}. s) \) for some \( o, s \in \mathcal{S} \)

\textbf{Let:} \( h \triangleq f(\langle e \rangle) \)

\( \langle 1 \rangle \) 1. \( \omega \odot f(\langle e \rangle) = f(\omega \odot \langle e \rangle) \)

\( \langle 2 \rangle \) 1. \( E_{l_o} \subseteq \omega \) for all \( l_o \in s \) and \( l_o \in o \)

\( \langle 3 \rangle \) 1. \( \omega = \bigcup_{l_s \in s, l_o \in o \in E_l} E_{l_s} \)

\textbf{Proof:} \( \omega = \text{RT}.o \cap \text{E}.s = \bigcup_{l_s \in s, l_o \in o} \text{RT}. \{l_o\} \cap \text{E}. \{l_s\} = \bigcup_{l_s \in s, l_o \in o \in E_l} E_{l_s} \quad \text{Q.E.D.} \)

\textbf{Proof:} \( E_{l_o} \subseteq \bigcup_{l_s \in s, l_o \in o \in E_l} E_{l_s} \) obviously holds for some \( l_s \in s \) and \( l_o \in o \), thus \( \langle 2 \rangle \) 1 holds by \( \langle 3 \rangle \) 1.

\( \langle 2 \rangle \) 2. \textbf{Case:} \( e \in E_{l_o} \) for some \( l_o \in s \) and \( l_o \in o \)

\( \langle 3 \rangle \) 1. \( (E_{l_s} \odot h = h) \Rightarrow (\omega \odot h = h) \)

\textbf{Proof:} \( E_{l_s} \odot h = h \) implies that \( h = \langle e \rangle \) or that \( h \) only contains events in \( E_{l_s} \). \( \omega \odot h = h \) trivially holds in the former case, and in the latter case because \( E_{l_s} \subseteq \omega \) by \( \langle 2 \rangle \) 1.

\( \langle 3 \rangle \) 2. \( E_{l_s} \odot f(\langle e \rangle) = f(E_{l_s} \odot \langle e \rangle) \)

\textbf{Proof:} Holds since \( f \in \mathcal{F} \) and all functions in \( \mathcal{F} \) must satisfy requirement (1) in Sect. 5.1 by definition of \( \mathcal{F} \).

\( \langle 3 \rangle \) 3. Q.E.D.

\textbf{Proof:}

\begin{align*}
E_{l_s} \odot f(\langle e \rangle) & = f(E_{l_s} \odot \langle e \rangle) \quad \text{[Holds by } \langle 3 \rangle \text{2]} \\
E_{l_o} \odot f(\langle e \rangle) & = f(\langle e \rangle) \quad \text{[Holds since } e \in E_{l_o} \text{ (} \langle 2 \rangle \text{2]}] \\
E_{l_o} \odot h & = h \quad \text{[Def. of } h \text{]} \\
The & = h \quad \text{[By } \langle 3 \rangle \text{1]} \\
\omega \odot f(\langle e \rangle) & = f(\langle e \rangle) \quad \text{[Def. of } h \text{]} \\
\omega \odot f(\langle e \rangle) & = f(\omega \odot \langle e \rangle) \quad \text{[e } \in \omega \text{ by } \langle 2 \rangle \text{1 and } \langle 2 \rangle \text{2]} \\
\end{align*}

\( \langle 2 \rangle \) 3. \textbf{Case:} \( e \notin E_{l_o} \) for all \( l_o \in s \) and \( l_o \in o \)

\( \langle 3 \rangle \) 1. \( (E_{l_s} \odot h = \langle e \rangle) \Rightarrow (\omega \odot h = \langle e \rangle) \)

\textbf{Proof:} It is easy to see that the implication holds if \( h = \langle e \rangle \). If \( h \neq \langle e \rangle \), then the left side of the implication requires that \( h \) does not contain any events in \( E_{l_s} \), but this entails that the right side of the implication holds because \( E_{l_s} \subseteq \omega \) by \( \langle 2 \rangle \) 1.

\( \langle 3 \rangle \) 2. \( E_{l_s} \odot f(\langle e \rangle) = f(E_{l_s} \odot \langle e \rangle) \)

\textbf{Proof:} Holds since \( f \in \mathcal{F} \) and all functions in \( \mathcal{F} \) must satisfy requirement (1) in Sect. 5.1 by definition of \( \mathcal{F} \).
(3)3. Q.E.D.
Proof:
\[ E_{l_1,l_2} \odot f(\langle e \rangle) = f(E_{l_1,l_2} \odot \langle e \rangle) \]  
\[ E_{l_1,l_2} \odot f(\langle e \rangle) = f(\langle e \rangle) \]  
\[ E_{l_1,l_2} \odot h = f(\langle e \rangle) \]  
\[ \omega \odot h = f(\langle e \rangle) \]  
\[ \omega \odot f(\langle e \rangle) = f(\langle e \rangle) \] 
\[ \omega \odot f(\langle e \rangle) = f(\omega \odot \langle e \rangle) \]  
\[ \omega \odot h = f(\langle e \rangle) \]  
\[ \omega \odot f(\langle e \rangle) = f(\omega \odot \langle e \rangle) \] 
\[ \omega \odot h = f(\langle e \rangle) \]  
(2)4. Q.E.D.
Proof: By (2)2 and (2)3.

B.4.3 Lemma 2.3

Assume: 1. \( f(\langle \rangle) = h \) where \( f \in F \)
Prove: \( h = \langle e \rangle \)

(1)1. \( E_{l_1,l_2} \odot h = h \) for arbitrary \( l_1,l_2 \in \mathcal{L} \).
Proof:
\[ E_{l_1,l_2} \odot f(\langle e \rangle) = f(E_{l_1,l_2} \odot \langle e \rangle) \]  
\[ E_{l_1,l_2} \odot f(\langle e \rangle) = f(\langle e \rangle) \]  
\[ E_{l_1,l_2} \odot h = h \]  
* Holds because \( f \) satisfies requirement (1) in Sect. 5.1.

(1)2. Q.E.D.
Assume: (a) \( h \neq \langle e \rangle \)
Prove: False.

(3)1. \( \exists l, l' \in \mathcal{L} : \exists h' \in \mathcal{H} : E_{l,l'} \odot h' \neq h' \)
Proof: Trivial.

(3)2. Q.E.D.
Proof: Select \( l \) such that \( l = l_a \) and \( h' \) such that \( h' = \langle e \rangle \), then (3)1 holds by (3)2.

(2)1. Q.E.D.
Proof: (3)1 contradicts (1)1, thus assumption (a) is false.
Chapter 10

Paper 2: Maintaining Information Flow Security under Refinement and Transformation
Maintaining Information Flow Security under Refinement and Transformation

Fredrik Seehusen\textsuperscript{1,2} and Ketil Stølen\textsuperscript{1,2}
\textsuperscript{1} SINTEF ICT, Norway
\{Fredrik.Seehusen, Ketil.Stolen\}@sintef.no
\textsuperscript{2} Department of Informatics, University of Oslo, Norway

September 28, 2006

Abstract

We address the problem of maintaining information flow security under refinement and transformation. To this end we define a schema for the specification of secure information flow properties and show that all security properties defined in the schema are preserved by a notion of refinement. Refinement is a process that requires human guidance and is in general not subject for automation. A transformation on the other hand, is an executable function mapping specifications to specifications. We define an interpretation of transformations and propose a condition under which transformations maintain security.

1 Introduction

We address the problem of maintaining information flow security during the process of making an abstract specification more concrete. This problem has received little attention, yet it is of relevance in any real-life scenario in which security analysis is carried out on the basis of a specification that abstracts away details of the full implementation. For example, it is of little help to know that Java code or some state machine specification is secure w.r.t. some property if validity of the property is not maintained by the compiler. Hence, we need means and a corresponding theory to ensure that the transformation from the abstract level to the more concrete level maintains the security of the abstract level.

Proving security once and for all is in general not possible. One reason for this is that the concrete level often includes peculiarities that do not have any abstract equivalent. Consequently security must be proven again at the concrete level to ensure that these additional peculiarities introduced via transformation do not violate security. Although additional verification is often needed at the concrete level, we still want to check and maintain security properties on the basis of the abstract specifications. There are three main reasons for this. First, analysis is in general more feasible at the abstract level since the concrete level may include too much detail to make analysis practical. Second, abstract specifications are in general more platform independent than concrete specifications.
This means that analysis results are more reusable at the abstract levels. Third, abstract specifications tend to be more understandable than concrete specifications, hence it is in general easier to specify and check security requirements at abstract levels as opposed to the concrete levels.

In this report we consider security in the sense of secure information flow properties (see e.g. [4, 5, 16]). The notion of secure information flow provides a way of specifying security requirements by selecting a set of observers, i.e. abstractions of system entities, and then restricting allowed flow of information between the observers.

The process of making an abstract specification more detailed is known as refinement, and the relationship between information flow security and refinement has been researched for a fairly long time. In 1989 it was shown by Jacob [13] that secure information flow properties in general are not preserved by the standard notion of refinement. It has later been observed that the problem originates in the inability of most specification languages to distinguish between underspecification and unpredictability\(^1\) [10, 14, 19]. We argue that this distinction is essential if secure information flow properties are to be preserved under refinement. To this end, both the standard notion of refinement and all secure information flow properties proposed in literature have to be redefined such that this distinction is taken into consideration. We show how to do this in a formalism similar to STAIRS [7, 8, 9] by defining a schema (based on [16]) for specifying secure information flow properties such that all properties defined in the schema are preserved by refinement.

Refinement is a relation on specifications that formalizes the process of stepwise development by the removal of underspecification. A transformation on the other hand is a computable function mapping specifications to specifications. For example, a compiler mapping a program to machine code is a kind of transformation. Currently, there is much ongoing work on transformation in relation to OMG’s standardization activities on MDA (Model Driven Architecture) [18], where transformations characterize the mapping of PIM’s (Platform Independent Model) to PSM’s (Platform Specific Model). Motivated by this we give a semantic interpretation of transformations and propose a condition under which transformations maintain security.

In summary, the main contributions of this report are: (1) the definition of a schema for specifying secure information flow properties that are preserved by the STAIRS notion of refinement. (2) The definition of a notion of secure transformation that preserves security properties defined in our schema.

This report is structured as follows: Sect. 2 formalizes a notion of system specification. Sect. 3 describes what is meant by secure information flow. In Sect. 4 we present the STAIRS notion of refinement and propose a schema for specifying secure information flow properties that are preserved by this notion of refinement. In Sect. 5, we discuss security maintaining transformations. Sect. 6 provides conclusions and related work. Proofs of all the results of this report are given in the appendix.

\(^1\)Also termed probabilistic non-determinism [19].
2 System Specifications

We model the input-output behavior of systems by finite sequences of events called traces. An event represents the transmission or the reception of a message. Formally, an event is a pair \((k, m)\) consisting of a kind \(k\) and a message \(m\). An event whose kind equals \(!\) represents the transmission of a message, whereas an event whose kind equals \(?\) represents the reception of a message. A message is a triple \((a_1, a_2, s)\) consisting of a transmitter \(a_1\), a receiver \(a_2\), and a signal \(s\) representing the message body. Both transmitters and receivers are referred to as agents, i.e. system entities such as objects or components.

**Definition 1** The semantics of a system specification, denoted \(\Phi\), is a prefix-closed set of traces. A set of traces \(A\) is prefix-closed iff

\[ t \in A \land t' \sqsubseteq t \Rightarrow t' \in A \]

where \(\sqsubseteq\) is the standard prefix ordering on sequences.

The reason why we require prefix-closure is that the definition of many secure information flow properties proposed in literature rely on this requirement \([16, 23]\).

In the sequel, we will for short write “specification” instead “the semantics of system specification” when it clear from the context what is meant.

### 2.1 Notational Convention

We define some notational conventions and standard operations. \(P(A)\) denotes the power set of \(A\) defined \(\{X | X \subseteq A\}\). \(A^*\) denotes the set of all finite sequences over the set \(A\). A sequence of events, i.e. a trace, is written \(\langle e_1, e_2, ..., e_n \rangle\). The empty trace, i.e. the trace with no events is written \(\langle \rangle\). The projection of a trace \(t\) on a set of events \(E\), written \(t|_E\), is obtained from \(t\) by deleting all elements not in \(E\).

Further notational conventions are listed in Table 1. Here the notion \(a \in A\) means that the set \(A\) is ranged over by \(a\).

---

### Table 1: Notational conventions

<table>
<thead>
<tr>
<th>Set</th>
<th>Definition</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a \in A)</td>
<td>(A)</td>
<td>Set of agents.</td>
</tr>
<tr>
<td>(o \in O)</td>
<td>(P(A))</td>
<td>Set of observers.</td>
</tr>
<tr>
<td>(s \in S)</td>
<td>(A \times A \times S)</td>
<td>Set of signals.</td>
</tr>
<tr>
<td>(m \in M)</td>
<td>(A \times A \times S)</td>
<td>Set of messages.</td>
</tr>
<tr>
<td>(e \in E)</td>
<td>({!, ?} \times M)</td>
<td>Set of events.</td>
</tr>
<tr>
<td>(h \in H)</td>
<td>(\emptyset)</td>
<td>Set of high-level events.</td>
</tr>
<tr>
<td>(l \in L)</td>
<td>(\emptyset)</td>
<td>Set of low-level events.</td>
</tr>
<tr>
<td>(t \in T)</td>
<td>(E^*)</td>
<td>Set of traces.</td>
</tr>
</tbody>
</table>
3 Information Flow Security

By secure information flow we understand a restriction on allowed flow of information between observers, i.e. sets of agents. Secure information flow can be described by a flow policy and a secure information flow predicate, referred to as a security predicate for short. The flow policy restricts information flow between observers, while the security predicate defines what is meant by information flow. Formally, a flow policy is a relation on observers

\[ \rightarrow \subseteq O \times O \]

where \((o_1, o_2) \in \rightarrow\) requires that there shall be no information flow from \(o_1\) to \(o_2\).

For simplicity, we will in the sequel assume a fixed flow policy \(\{(H, L)\}\) consisting of two observers only: \(H\), the high-level observer and \(L\), the low-level observer.

Security predicates that describe what is meant by information flow are expressed in terms of the observations that observers can make. Formally, the observation that an observer \(o\) can make of a trace \(t\) is obtained from \(t\) by deleting all events that cannot be observed by \(o\):

\[ t|_{(e.o)} \]

Here \(e.o\) yields the set of all events that can be observed by \(o\), i.e. all events that can be transmitted from or received by the agents in \(o\):

\[ e.o = \{(k, (a, a', m)) \in E | (k=! \land a \in o) \lor (k=? \land a' \in o)\} \]

For short, we let \(L\) and \(H\) denote \(e.L\) and \(e.H\), respectively.

To ensure that \(L\) cannot observe high-level events directly, we demand that \(H \cap L = \emptyset\). This alone does not in general prevent information flow from \(H\) to \(L\) because \(L\) may infer confidential information from \(H\) based on the observations that \(L\) can make. A central notion in defining what is meant by inferences is that of low-level indistinguishability.

**Definition 2** Two traces \(t\) and \(t'\) are indistinguishable from \(L\)'s point of view, written \(t \sim_L t'\), iff

\[ t|_{L} = t'|_{L} \]

That is, iff \(L\)'s observation of \(t\) is equal to \(L\)'s observation of \(t'\).

In the sequel we assume that \(L\) has complete knowledge of the specification \(\Phi\) that describes the set of all possible behaviors represented by traces. This means that \(L\) may construct the set of all traces in \(\Phi\) that are indistinguishable or compatible with a given observation. Formally, \(L\) may from the observation of any trace \(t\), construct a so-called low-level equivalence set [23] (abbreviated LLES in the sequel):

\[ \{t' \in \Phi | t \sim_L t'\} \]

In other words, if \(L\) makes an observation \(t\), then \(L\) can infer that some trace in the LLES constructed from \(t\) has occurred, but not which one. Security predicates must demand that \(L\) shall not be able to deduce confidential information from the LLESs that \(L\) may construct. This is illustrated in the following example.
Example Let $\Phi = \{\langle\rangle, \langle l_1 \rangle, \langle h_1 \rangle, \langle h_2 \rangle, \langle h_1, l_1 \rangle, \langle h_1, l_2 \rangle, \langle h_2, l_2 \rangle\}$, and assume that $L$ may observe events $l_1$ and $l_2$ and that $h_1$ and $h_2$ are high-level events. Assume further a definition of security that states that $L$ shall not with certainty be able to infer that a high-level event has occurred. If $L$ makes the observation $\langle l_1 \rangle$, then $L$ may infer that either trace $\langle l_1 \rangle$ or trace $\langle h_1, l_1 \rangle$ have occurred. Since $L$ cannot know which of these has occurred (because $L$ cannot observe high-level events directly), and the former trace does not contain any high-level events, $L$ cannot infer with certainty that a high-level event has occurred. If $L$ on the other hand observes the trace $\langle l_2 \rangle$, then $L$ can infer that $\langle h_1, l_2 \rangle$ or $\langle h_2, l_2 \rangle$ have occurred. Again, $L$ does not know which of these has occurred, but since both traces contain a high-level event and there are no other traces in $\Phi$ that are compatible with $L$’s observation, $L$ can infer with certainty that a high-level event has occurred. Hence, $\Phi$ is not secure w.r.t. our definition of security.

In order for $\Phi$ to be secure, one must demand that the LLESSs that $L$ can construct be closed w.r.t. some criterion [16]. In the above example, this amounts to demanding that there must be a trace with no high-level events in each LLES that $L$ can construct.

3.1 Mantel’s Assembly Kit

The schema we propose for describing security predicates is based on a schema proposed by Mantel [16]. He presents an assembly kit in which different notions of security can be defined. We give here a brief description of this assembly kit. The reader is referred to [16] for a more details.

In Mantel’s framework, security properties are represented as security predicates where a security predicate $Sp$ is either a single basic security predicate $Bsp$, or a conjunction of basic security predicates. Each basic security predicate $Bsp$ demands that for any trace $t$ of the specification $\Phi$ there must be another trace $t'$ that is indistinguishable from $t$ from $L$’s point of view, and which fulfills a condition $Q$, the closure requirement of $Bsp$. The existence of $t'$, however, is only required if a condition $R$, the restriction of $Bsp$, holds. This results in the following schema for the formal definition of basic security predicates:

$$
\text{Definition 3} \quad \text{Specification } \Phi \text{ satisfies the basic security predicate } Bsp_{QR}(\Phi) \text{ for restriction } R \text{ and closure requirement } Q \text{ iff } \forall t \in \Phi \cdot R(\Phi, t) \Rightarrow \exists t' \in \Phi \cdot t \sim L t' \land Q(t, t')
$$

Example The notion of security that is informally described in Ex. 3, may be defined by instantiating the schema as follows: $R \triangleq \text{true}$, and $Q \triangleq t'|_{H} = \langle \rangle$. That is, for every trace $t$ there must be a trace $t'$ such that $t'$ is indistinguishable from $t$ w.r.t. $L$ and such that $t'$ does not contain any high-level events.

4 Refinement

Refinement is the process of making an abstract specification more concrete by removing underspecification. The standard notion of refinement [11] states that
a system specification $\Phi'$ is a refinement of a system specification $\Phi$ iff

$$
\Phi' \subseteq \Phi
$$

Intuitively, there are at least as many implementations that satisfy $\Phi$ as there are implementations that satisfy $\Phi'$. In this sense $\Phi'$ describes its set of implementations more accurately than $\Phi$, ergo $\Phi'$ is less abstract than $\Phi$.

The reason why secure information flow properties are not preserved by refinement becomes apparent when one considers again the manner in which these properties are defined (see Def. 3). That is, $\Phi$ is secure if some of its traces satisfy the closure requirement $Q$. However, by (2) there is no guarantee that a refinement of $\Phi$ will include those traces that satisfy $Q$, hence secure information flow properties are in general not preserved by refinement.

Intuitively, the cause of this problem is that security properties depend on unpredictability. E.g. the strength of one’s password may be measured in terms of how hard it is for an attacker to guess the password one has chosen. The closure requirement $Q$ may be seen as the security predicate’s requirement of unpredictability, but traces that provide this unpredictability may be removed during refinement. This motivates a redefinition of the notions of specification and refinement where the distinction between underspecification and unpredictability is taken into consideration.

**Definition 4** A system specification, denoted $\Omega$, is a set of trace sets. Each trace set in a specification is called an obligation. We demand that the set obtained by collapsing $\Omega$ into a set of traces must be prefix-closed, i.e. we demand

$$
t \in \hat{\Omega} \land t' \sqsubseteq t \Rightarrow t' \in \hat{\Omega}
$$

where $\hat{\Omega}$ is defined $\bigcup_{\phi \in \Omega} \phi$.

**Definition 5** System specification $\Omega'$ is a refinement of system specification $\Omega$, written $\Omega \rightarrow \Omega'$, iff

$$(\forall \phi \in \Omega \cdot \exists \phi' \in \Omega' \cdot \phi' \subseteq \phi) \land (\forall \phi' \in \Omega' \cdot \exists \phi \in \Omega \cdot \phi' \subseteq \phi)$$

This corresponds to so-called limited refinement in STAIRS [20]. For an arbitrary obligation $\phi$ at the abstract level, there must be an obligation $\phi'$ at the concrete level such that $\phi'$ is a refinement of $\phi$ in the sense of the standard notion of refinement (see (2)). Moreover, each obligation at the concrete level must be a refinement of an obligation at the abstract level. The latter ensures that behavior that was not considered at the abstract level is not introduced at the concrete level.

The intuition is that the traces within the same obligation may provide underspecification, while the obligations provide unpredictability in the sense that an implementation is required to fulfill all obligations of a specification. Any valid implementation must potentially exhibit the behavior described by at least one trace in each obligation. By implementation we understand a specification with no underspecification. Given some program $P$, let $\text{Traces}(P)$ be the prefix-closed set of traces that can be generated by executing $P$, and let

$$
[ P ] \triangleq \{ \{ t \} \mid t \in \text{Traces}(P) \}$$
Then $P$ implements specification $\Omega$ iff

$$\Omega \leadsto [P]$$

**Example** Let $\Omega = \{\{\}\}, \{(l)\}, \{(l), (h_1)\}, \{(l), (h_2)\}, \{(h_1, l)\}, \{(h_2, l)\}\}$, and assume that it is confidential that high-level events have occurred. $\Omega$ is secure in this respect; it is easy to verify that this holds for all implementations of $\Omega$.

**Lemma 1** $\leadsto$ is transitive:

$$\Omega \leadsto \Omega' \land \Omega' \leadsto \Omega'' \Rightarrow \Omega \leadsto \Omega''$$

Instances of the schema of Def. 3 are in general not preserved by our notion of refinement. We need to modify the schema such that the distinction of unpredictability and underspecification is exploited. Instead of demanding that there is a trace $t'$ that satisfies some criterion, we demand that there is an obligation $\phi$ such that all its traces satisfy that criterion.

**Definition 6** Specification $\Omega$ satisfies the basic security predicate $\text{Bsp}_{QR}(\Omega)$ for restriction $R$ and closure requirement $Q$ iff

$$\forall t \in \hat{\Omega} \cdot R(\hat{\Omega}, t) \Rightarrow \exists \phi \in \Omega \cdot \forall t' \in \phi \cdot t \sim t' \land Q(t, t')$$

The intuition of (Def. 6) is that obligations, as opposed to individual traces, may be seen as providing the unpredictability required by instances of the schema. Note that the schema may be instantiated by the same instances of $R$ and $Q$ that are presented in Mantel’s paper.

In order to ensure that instances of the schema are preserved by refinement, we need to disallow some restrictions $R$ whose truth value depend on the absence of traces. We therefore require that all restrictions $R$ satisfy the following condition

$$(T' \subseteq T \land R(T', t)) \Rightarrow R(T, t) \quad (3)$$

for arbitrary traces $t$ and trace sets $T$ and $T'$. All instances of $R$ presented in Mantel’s paper satisfy condition (3).

**Theorem 1** $\text{Bsp}_{QR}$ is preserved by refinement for arbitrary restrictions $R$ satisfying (3) and closure requirements $Q$:

$$\Omega \leadsto \Omega' \land \text{Bsp}_{QR}(\Omega) \Rightarrow \text{Bsp}_{QR}(\Omega')$$

The notion of refinement introduced above corresponds to what is often referred to as property refinement or behavioral refinement [2]. Property refinement does not capture change in data-structure, i.e. the replacement of abstract event representations by concrete event representations. This is in contrast to refinement notions such as data refinement [12], interface refinement [2], or action refinement [22] which roughly speaking may be understood as property refinement modulo a translation between the concrete and the abstract data structure. Our notion of property refinement may be generalized into a notion of action refinement (actually event refinement in our case) using upwards and downwards simulation [3, 12] in a fairly standard manner. To characterize under which conditions this notion of refinement is security preserving is, however, far from trivial. In the following, attention is restricted to a special case of this problem, namely under which conditions transformations are security preserving. A transformation may be understood a special case of action refinement where the concrete specification is generated automatically from the abstract specification.
5 Transformation

The notion of refinement addressed above is a binary relation on specifications that formalizes the process of stepwise development by removal of underspecification. A transformation on the other hand is an executable function taking an abstract syntactic specification as input and yielding a concrete syntactic specification as output. Thus transformation is a syntactic notion. Since security properties are defined on the semantics of specifications (i.e. on traces), we define a semantic interpretation of transformations which enables us to assert whether a transformation maintains security.

In Sect. 5.1, we give an example of a transformation that motivates our semantic interpretation of transformations given in Sect. 5.2. In Sect. 5.3, we propose a condition under which interpretations of transformations maintain security. Sect. 5.4 gives an example that clarifies some of the points made in Sect. 5.3.

5.1 Example: Transforming HTTP Specifications to TCP Specifications

The HTTP protocol is bound to the TCP protocol. One way of doing this binding during runtime is to create a new so-called TCP session for each HTTP request-response pair. A TCP-session consists of three phases: First a connection is established between the two sides of communication, then the HTTP request and response messages are segmented, encapsulated by TCP frames, and transmitted. Finally the connection is explicitly terminated.

A transformation that takes a specification that describes communication at the HTTP level and produces a specification that describes communication at the TCP level may be defined in accordance to how the HTTP protocol is bound to the TCP protocol. Such a transformation may be regarded as a transformation from the abstract to the concrete if one takes HTTP specifications as being at the abstract level and TCP specifications as being at the concrete level.

The UML interaction diagram on the left hand side of Fig. 1 describes a simple communication scenario at the HTTP level. The diagram on the right hand side is the result of applying a transformation from HTTP to TCP to the HTTP diagram. Here the so-called xalt-operator from STAIRS [8] speci-
TRANSFORMATION

5  TRANSFORMATION

fies unpredictability\(^2\) between different interaction scenarios. The ref-operator references other interaction diagrams that in this example describe different TCP-sessions. The reason why the HTTP request-response pair described in the diagram on the left hand side is translated to more than one TCP-session is that the TCP protocol must handle issues that are transparent at the HTTP level, e.g. message overtaking. One TCP session may for example describe the situation in which messages are received in the same order that they are transmitted, another may describe the situation in which this is not the case and so on. The reader is referred to [7, 8, 9, 21] to see how UML interaction diagrams can be given trace semantics.

To assert whether the transformation from HTTP to TCP maintains security, we need to interpret the transformation in terms of how abstract traces (representing HTTP communication) are translated to concrete traces (representing TCP communication). There are three considerations that need to be taken into account when defining such an interpretation. First, an abstract trace may correspond to several concrete traces. The reasons for this is that TCP protocol must handle issues that are transparent at the HTTP level. Second, an abstract event may be decomposed into several concrete events because each HTTP package may be segmented into more than one TCP package during the TCP transmission phase. Third, there may be traces at the concrete level that have no abstract equivalent. To see this, let \(\langle e \rangle\) represent the transmission of a HTTP package. Since a HTTP package may be segmented into several TCP packages, \(\langle e \rangle\) may for example be translated into the trace \(\langle e_1, e_2, e_3 \rangle\) where events \(e_1, e_2,\) and \(e_3\) represent the transmission of TCP packages. Traces \(\langle e_1 \rangle\) and \(\langle e_1, e_2 \rangle\) may also be valid traces at the TCP level (these traces are in fact required to be present in a concrete specification since we assume that specifications are prefix-closed). Now, the TCP trace \(\langle e_1, e_2, e_3 \rangle\) corresponds to \(\langle e \rangle\) at the HTTP level since the TCP trace is complete in the sense that it represents the transmission of the entire HTTP message. But what about the TCP traces \(\langle e_1 \rangle\) and \(\langle e_1, e_2 \rangle\), do these traces correspond to \(\langle e \rangle\) at the abstract level? The answer is no if the trace \(\langle e \rangle\) is meant to represent the transmission of an entire HTTP package. The TCP traces \(\langle e_1 \rangle\) and \(\langle e_1, e_2 \rangle\) do not correspond to the empty HTTP trace \(\langle \rangle\) either, because the empty trace is meant (by any reasonable interpretation) to represent a scenario in which no communication occurs. From this we can conclude that, in general, there may be traces at the concrete level for which there are no corresponding traces at the abstract level.

5.2 Transformations from a Semantic Perspective

Syntactically, a transformation is an executable function translating abstract (syntactic) specifications to concrete (syntactic) specifications. Semantically, we interpret traces in terms of how abstract traces are translated to concrete traces.

In the following, let \(A\) denote some fixed but arbitrary abstract syntactic specification, \(T\) be a some transformation, and \(T(A)\) denote the concrete specification obtained by applying \(T\) to \(A\). Let \(\Omega_a\) denote the semantics of \(A\) and \(\Omega_c\) denote the semantics of \(T(A)\). \(T\) is interpreted by a set of functions

\[
F \subseteq \hat{\Omega}_a \rightarrow \hat{\Omega}_c
\]

\(^2\)Termed explicit non-deterministic choice in STAIRS
mapping traces in $\Omega_a$ to traces in $\Omega_c$. The reason why we use a set of functions and not a single function is, as explained in the example, that a syntactic transformation may represent the same abstract trace by several concrete traces.

We say that the set of functions $F$ is a valid interpretation of $T$ w.r.t. $A$ if $\Omega_c$ is a translation of $\Omega_a$ w.r.t. $F$ as defined below. We first define the notion of translation for obligations, then we lift this notion to (semantic) specifications.

**Definition 7** Obligation $\phi_c$ is a translation of obligation $\phi_a$ w.r.t. function $f$, written $\phi_a \hookrightarrow_f \phi_c$, iff

$$\phi_c \subseteq \{ f(t) \mid t \in \phi_a \}$$

**Definition 8** Specification $\Omega_c$ is a translation of specification $\Omega_a$ w.r.t. interpretation $F$, written $\Omega_a \hookrightarrow_F \Omega_c$, iff

$$\forall f \in F \cdot \forall \phi_a \in \Omega_a \cdot \exists \phi_c \in \Omega_c \cdot \phi_a \hookrightarrow f \phi_c$$

Our interpretation of transformations is similar to data refinement in that both notions roughly speaking may be understood as refinement modulo a translation of traces. More precisely:

**Lemma 2** Let $\Omega_c$ be contained in the image of $\Omega_a$ under the identity transformation $\text{id}$, then

$$\Omega_a \hookrightarrow_{\text{id}} \Omega_c \Leftrightarrow \Omega_a \hookrightarrow \Omega_c$$

Here $\text{im}(\Omega_a, F)$, the image of $\Omega_a$ under some $F$, is the set of obligations that are translations of obligations in $\Omega_a$ w.r.t. $F$:

$$\text{im}(\Omega_a, F) \triangleq \{ \phi_c \mid \exists \phi_a \in \Omega_a \cdot \exists f \in F \cdot \phi_a \hookrightarrow f \phi_c \}$$

(4)

A concrete specification is not necessarily contained in the image of the abstract specification it is translated from. The reason for this is, as explained in the previous example, that there may be concrete traces that do not have any abstract equivalent.

If $F_1$ and $F_2$ are interpretations, then $F_1 \circ F_2$ is understood as the interpretation obtained by functional point-to-point composition of all functions from $F_1$ and $F_2$. That is,

$$F_1 \circ F_2 \triangleq \{ f_1 \circ f_2 \mid f_1 \in F_1 \land f_2 \in F_2 \}$$

(5)

where $f_1 \circ f_2(t) = f_1(f_2(t))$.

**Lemma 3** $\hookrightarrow$ is transitive:

$$\Omega_0 \hookrightarrow_{F_0} \Omega_1 \land \Omega_1 \hookrightarrow_{F_2} \Omega_2 \Rightarrow \Omega_0 \hookrightarrow_{F_2 \circ F_1} \Omega_2$$

We denote by $F^{-1}(t_c)$, the set of all traces in $\hat{\Omega}_a$ that can be translated to $t_c$ by the functions in $F$:

$$F^{-1}(t_c) \triangleq \{ t_a \in \hat{\Omega}_a \mid \exists f \in F \cdot f(t_a) = t_c \}$$

(6)
5.3 Secure Transformations

A transformation $T$ maintains the security of an abstract specification $A$ if there is a valid secure interpretation of $T$ w.r.t. $A$. The notion of secure interpretation obviously depends on what is meant by secure, and should therefore be parameterized by security properties. In doing so, one must take into account that the security requirement at the abstract level may be syntactically and semantically different from the “corresponding” security requirement at the concrete level. One reason for this is that events at the abstract level may differ from the events at the concrete level. Another reason is that there may be concrete traces that do not have any corresponding abstract trace. The concrete security requirement must therefore handle traces that may not be taken into account by the security requirement at the abstract level.

The notion of secure interpretation is formally defined in the following Definition 9.

**Definition 9** Let $\Omega_a \hookrightarrow F \Omega_c$, then the interpretation $F$ is secure w.r.t. the abstract and concrete restrictions $R_a$ and $R_c$ and abstract and concrete closure requirements $Q_a$ and $Q_c$, if the following conditions are satisfied

$$R_c(\hat{\Omega}_c, t_c) \Rightarrow \exists t' \in F^{-1}(t_c) \cdot R_a(\hat{\Omega}_a, t')$$  \hspace{1cm} (7)

$$(R_c(\hat{\Omega}_c, t_c) \land \forall t' \in \phi_a \cdot t_a \sim t' \land Q_a(t_a, t')) \Rightarrow \exists \phi_c \in \Omega_c \cdot \forall t' \in \phi_c \cdot t_c \sim t' \land Q_c(t_c, t')$$  \hspace{1cm} (8)

for all $t_c \in \hat{\Omega}_c$, $\phi_a \in \Omega_a$, and $t_a \in F^{-1}(t_c)$.

Def. 9 may be understood to capture a rule that allows us to exploit that we have established $Q_a$ at the abstract level when establishing $Q_c$ at the concrete level. We believe that verifying (7) and (8) in most practical situations will be straightforward and more feasible than checking the security property at the concrete level directly.

Put simply, the first condition of Def. 9 just ensures that the transformation does not weaken the restriction $R$. The second condition ensures that the transformation does not strengthen the closure requirement $Q$ and that low-level equality is preserved.

It follows from (7) that the concrete restriction $R_c$ must filter away (i.e. yield false for) the concrete traces $t_c$ that do not have any corresponding abstract trace. This is reasonable because one cannot take advantage of the fact that the abstract specification is secure when proving that $t_c$ does not compromise security. It may therefore be the case that a new security analysis must be carried out at the concrete level for those traces that do not have an abstract equivalent.

When we relate Def. 9 to rules that describe date refinement in an assumption / guarantee or pre-post setting, we note that weakening/strengthening is the other way around. E.g., when refining a pre-post specification, one may weaken the pre-condition and strengthen the post-condition. The reason is that a pre-post condition is a specification that is refined into another specification while a restriction-closure predicate is a property that has been proved to hold for a specification that is translated to a concrete specification and whose validity should be maintained.
Theorem 2 Let $F$ be an interpretation that is secure w.r.t. restrictions $R_a$ and $R_c$, and closure requirements $Q_a$ and $Q_c$, then $F$ maintains security in the following sense:

$$\text{Bsp}_{Q_a R_a}(\Omega_a) \land \Omega_a \hookrightarrow F \Omega_c \Rightarrow \text{Bsp}_{Q_c R_c}(\Omega_c)$$

5.4 Example: Why security requirements change

Let $\Omega_a$ be an abstract specification consisting of two clients $c_l$ and $c_h$ that communicate with a server $s$ via the HTTP protocol (see Fig. 2). Assume that $c_l$, based on its observation of its communication with $s$ and its knowledge of the system specification, should not be able to deduce information about the behavior of $c_h$. More formally, both the clients and the server can be represented as agents (recall the role of agents from Sect. 2). Thus the low-level observer is defined $\{c_l\}$ and the high-level observer is defined $\{c_h\}$. The security requirement on the HTTP level may be formalized by instantiating the schema of Def. 6 by some predicates $R_a$ and $Q_a$.

Assume that $F$ interprets a transformation from HTTP to TCP defined such that each event representing the transmission or reception of a HTTP message is translated into a complete (in the sense of the previous example) sequence of events describing the corresponding TCP messages. Let $\Omega_c$ be a translation of $\Omega_a$ w.r.t. $F$ and assume that $\Omega_c$ also contains non-complete traces that do not correspond to any traces at the HTTP level.

Assume that $\Omega_a$ is secure (i.e. $\text{Bsp}_{R_a Q_a}(\Omega_a)$ is true) and that we want to check if the traces on the concrete level that have a corresponding abstract representation are secure w.r.t. our information flow property. In order to do this, we can create predicate on the concrete level that filters away those traces that do not have a corresponding abstract representation. More formally, the concrete restriction $R_c$ is defined $R_c(\hat{\Omega}, t) \equiv R_a(\hat{\Omega}, t) \land \text{TCP}_{OK}(t)$ where $\text{TCP}_{OK}$ is a predicate that yields true if the TCP trace $t$ corresponds to a HTTP trace and false otherwise. Note that we are assuming that $R_a$ may take HTTP traces as well as TCP traces as arguments. The concrete security property can now be obtained by instantiating the schema of Def. 6 by the predicates $R_c$ and $Q_a$ (since the closure requirement $Q$ is left unchanged in this example).

If one wants to check that the TCP traces that do not correspond to any HTTP traces are secure w.r.t. some requirement, one cannot take advantage of the fact that $\Omega_a$ is secure. Therefore additional security analysis may be required w.r.t. these traces.
6 Conclusions and Related Work

In [21], we defined a secure information flow property in the semantics of STAIRS [7, 8, 9] and showed that this property was preserved by refinement and transformation. This report simplifies and generalizes these results by considering, not only one, but many kinds of information flow properties. This report also considers a more general notion of security preservation than considered in [21]. More precisely, this report makes two contributions to the study of secure information flow. The first is a schema for specifying information flow properties that are preserved by the STAIRS notion of refinement. The second is the definition of a semantic interpretation of transformations and a condition under which transformations maintain security.

There are a number of papers related to information flow security and refinement. Jacob is the first person that we are aware of to show that secure information flow properties are not preserved by the traditional notion of refinement [13]. This became known as the refinement paradox. It has later been observed that this “paradox” is a manifestation of failing to clearly distinguish between underspecification and unpredictability. As far as we are aware of, this observation was first made in [19].

To the extent of our knowledge, the work of Heisel. et. al. [10] is similar to ours in that they both distinguish between underspecification and unpredictability and consider the notion of data refinement. The main differences between their work and ours are: (1) They work in a probabilistic setting, and thus their formalism differs from ours. (2) They do not consider information flow properties but a notion of confidentiality based on low-level indistinguishability only. (3) Their notion of confidentiality preserving refinement is different from ours in that they build the condition of confidentiality preservation into the definition of refinement. W.r.t. refinement, we have taken the dual approach of strengthening the notion of security.

The work of J"urjens [14, 15] is also related to ours. Some of the main differences between his work and ours are: (1) His formalism differs from ours. (2) While J"urjens distinguishes between underspecification and unpredictability in order to define refinement preserving properties of confidentiality (secrecy) and integrity, he does not rely on this distinction in the definition of his information flow property. That is, the secure information flow property is satisfied iff each behavior refinement to a deterministic specification satisfies this property, i.e. he effectively closes the property under a notion of behavior refinement that does not distinguish between underspecification and unpredictability.

Three notable papers that addresses information flow security and refinement are [1, 6, 17]. The main difference between these papers and ours is that all these investigate conditions under which certain notions of refinement are security preserving without distinguishing between underspecification and unpredictability. Since this distinction is made in our formalism, we consider one notion of refinement only, and strengthen instead our notion of security in an intuitive manner. Hence, there is no need to propose conditions with which to check that a given refinements preserve security.

We are not aware of any work that explicitly address transformation and information flow security. Moreover, we are not aware of any work that show how to preserve secure information flow properties under a notion of refinement that takes the distinction of underspecification and unpredictability into
consideration.

The main emphasis of this report is on semantics. In future work, we will address syntactic transformations in more detail. We are also planning to address composition and transformation of security predicates. Eventually, we would like to develop a computerized tool that will check whether transformations maintain security.

Acknowledgments

This work has been funded by the Research Council of Norway through the project SECURIS (152839/220). We would like to thank the anonymous referees for their useful comments and suggestions.

References


REFERENCES


A Proofs

A.1 Lemma 1

\( \rightarrow \) is transitive.

Assume:
1. \( \Omega \rightarrow \Omega' \)
2. \( \Omega' \rightarrow \Omega'' \)

Prove: \( \Omega \rightarrow \Omega'' \)

\(1.1\) \( (\forall \phi \in \Omega' \cdot \exists \phi'' \in \Omega'' \cdot \phi'' \subseteq \phi) \land (\forall \phi'' \in \Omega'' \cdot \exists \phi \in \Omega \cdot \phi \subseteq \phi) \)

\(1.2\) \( (\forall \phi \in \Omega \cdot \exists \phi'' \in \Omega'' \cdot \phi'' \subseteq \phi) \)

\(3.1\) \( \exists \phi'' \in \Omega'' \cdot \phi'' \subseteq \phi \) for arbitrary \( \phi \in \Omega \)

\(4.1\) Choose \( \phi' \in \Omega' \) and \( \phi'' \in \Omega'' \) such that \( \phi' \subseteq \phi \) and \( \phi'' \subseteq \phi' \)

Proof: By assumptions 1 and 2 and definition of \( \rightarrow \) (Def. 5).

\(4.2\) \( \phi'' \subseteq \phi \)

Proof: By \(4\)1 and transitivity of \(\subseteq\).

\(4.3\) Q.E.D.

Proof: By \(4\)1 and \(4\)2.

\(3.2\) Q.E.D.

Proof: \(\forall\)-rule.

\(2.1\) \( (\forall \phi' \in \Omega' \cdot \exists \phi \in \Omega \cdot \phi' \subseteq \phi) \)

\(3.1\) \( \exists \phi \in \Omega \cdot \phi'' \subseteq \phi \) for arbitrary \( \phi'' \in \Omega'' \)

\(4.1\) Choose \( \phi' \in \Omega' \) and \( \phi \in \Omega \) such that \( \phi'' \subseteq \phi' \) and \( \phi' \subseteq \phi \)

Proof: By assumptions 1 and 2 and definition of \( \rightarrow \) (Def. 5).

\(4.2\) \( \phi'' \subseteq \phi \)

Proof: By \(4\)1 and transitivity of \(\subseteq\).

\(4.3\) Q.E.D.

Proof: By \(4\)1 and \(4\)2.

\(3.2\) Q.E.D.

Proof: \(\forall\)-rule.

\(2.3\) Q.E.D.

Proof: By \(2\)1 and \(2\)2.

\(1.2\) Q.E.D.

Proof: By \(1\)1 and definition of \( \rightarrow \) (Def. 5).

A.2 Theorem 1

Bsp\(QR\) is preserved by refinement for arbitrary restrictions \( R \) satisfying \(3\) and closure requirements \( Q \).

Assume:
1. \( (T' \subseteq T \land R(T', t)) \Rightarrow R(T, t) \) for arbitrary \( T, T' \) and \( t \). 
2. \( \Omega \rightarrow \Omega' \)
3. Bsp\(QR\)(\(\Omega\))

Prove: Bsp\(QR\)(\(\Omega'\))

\(1.1\) Assume: 1.1 \( t \in \hat{\Omega'} \)
1.2 \( R(\hat{\Omega'}, t) \)
PROVE: \( \exists \phi' \in \Omega' \cdot \forall t' \in \phi' : t \sim t' \land Q(t', t) \)

(2.1) \( R(\tilde{\Omega}, t) \)

(3.1) \( \tilde{\Omega}' \subseteq \tilde{\Omega} \)

Proof: By assumption 2, definition of \( \sim \) (Def. 5), and definition of \( \sim \) (Def. 4).

(3.2) Q.E.D.

Proof: By assumption 1, (3.1), and assumption 1.

(2.2) Choose \( \phi \in \Omega \) such that \( t \sim t' \) and \( Q(t, t') \) for arbitrary \( t' \in \phi \)

Proof: By (2.1), assumption 3 and definition of BSP\(_{QR} \) (Def. 6).

(2.3) Choose \( \phi' \in \Omega' \) such that \( \phi' \subseteq \phi \)

Proof: By assumption 2, (2.2) and definition of \( \sim \) (Def. 5).

(2.4) \( t \sim t' \) and \( Q(t', t) \) for arbitrary \( t' \in \phi' \)

Proof: By (2.2) and (2.3).

(2.5) Q.E.D.

Proof: By (2.3) and (2.4).

(1.2) Q.E.D.

Proof: By (1.1) and definition of BSP\(_{QR} \) (Def. 6).

A.3 Lemma 2

Let \( \Omega_c \) be contained in the image of \( \Omega_a \) under the identity transformation \( id \), then

\[ \Omega_a \hookrightarrow id \Omega_c \Leftrightarrow \Omega_a \sim \Omega_c \]

Proof: By lemma 2.1 and 2.2.

A.3.1 Lemma 2.1

Assume:

1. \( id \in \tilde{\Omega}_a \rightarrow \tilde{\Omega}_c \), and \( id(t) = t \) for all \( t \in \tilde{\Omega}_a \)

2. \( \Omega_c \subseteq \text{im}(\Omega_a, \{id\}) \)

3. \( \Omega_a \hookrightarrow id \Omega_c \)

Proof:

(1.1) \( (\forall \phi_a \in \Omega_a \cdot \exists \phi_c \in \Omega_c \cdot \phi_c \subseteq \phi_a) \land (\forall \phi_c \in \Omega_c \cdot \exists \phi_a \in \Omega_a \cdot \phi_a \subseteq \phi_c) \)

(2.1) \( \forall \phi_a \in \Omega_a \cdot \exists \phi_c \in \Omega_c \cdot \phi_c \subseteq \phi_a \)

(3.1) \( \exists \phi_c \in \Omega_c \cdot \phi_c \subseteq \phi_a \) for arbitrary \( \phi_a \in \Omega_a \)

(4.1) Choose \( \phi_c \in \Omega_c \) such that \( \phi_a \hookrightarrow id \phi_c \)

Proof: By assumption 3 and definition of \( \hookrightarrow \) (Def. 8).

(4.2) \( \phi_c \subseteq \phi_a \)

Proof: By (4.1), assumption 1, and definition of \( \hookrightarrow \) (Def. 7).

(4.3) Q.E.D.

Proof: By (4.1) and (4.2).

(3.2) Q.E.D.

Proof: By \( \forall \)-rule.

(2.2) \( \forall \phi_c \in \Omega_c \cdot \exists \phi_a \in \Omega_a \cdot \phi_c \subseteq \phi_a \)

(3.1) \( \exists \phi_a \in \Omega_a \cdot \phi_a \subseteq \phi_c \) for arbitrary \( \phi_c \in \Omega_c \)

(4.1) Choose \( \phi_a \in \Omega_a \) such that \( \phi_a \hookrightarrow id \phi_c \)

Proof: By assumptions 2 and 3, and definition of \( \text{im} \) (4) and \( \hookrightarrow \) (Def. 8).

(4.2) \( \phi_c \subseteq \phi_a \)
A PROOFS

Proof: By (4)1, assumption 1, and definition of $\rightsquigarrow$ (Def. 7).

(4)3. Q.E.D.

Proof: By (4)1 and (4)2.

(3)2. Q.E.D.

Proof: By $\forall$-rule.

(2)3. Q.E.D.

Proof: By (2)1 and (2)2.

(1)2. Q.E.D.

Proof: By (1)1 and definition of $\Rightarrow$ (Def. 5).

A.3.2 Lemma 2.2

Assume:
1. $\Omega_a \rightsquigarrow \Omega_c$
2. $id \in \widehat{\Omega}_a \Rightarrow \widehat{\Omega}_c$ and $id(t) = t$ for all $t \in \widehat{\Omega}_a$

Prove:
$\Omega_a \leftarrow (id) \Omega_c$

(1)1. $\forall \phi_a \in \Omega_a \cdot \exists \phi_c \in \Omega_c \cdot \phi_a \leftarrow (id) \phi_c$
(2)1. $\exists \phi_a \in \Omega_c \cdot \phi_a \leftarrow (id) \phi_c$ for arbitrary $\phi_a \in \Omega_a$
(3)1. Choose $\phi_c \in \Omega_c$ such that $\phi_c \subseteq \phi_a$
    Proof: By assumption 1 and definition of $\leftarrow$ (Def. 5).
(3)2. $\phi_a \leftarrow (id) \phi_c$
    Proof: By (3)1, assumption 2, and definition of $\leftarrow$ (Def. 7).
(3)3. Q.E.D.
    Proof: By (3)1 and (3)2.
(2)2. Q.E.D.
    Proof: By $\forall$-rule.
(1)2. Q.E.D.
    Proof: By (1)1 and definition of $\leftarrow$ (Def. 8).

A.4 Lemma 3

$\leftarrow$ (Def. 8) is transitive.

Assume:
1. $\Omega \leftarrow_{F_1} \Omega_1$ for $F_1 \in \mathcal{F}$
2. $\Omega_1 \leftarrow_{F_2} \Omega_2$ for $F_2 \in \mathcal{F}$

Prove:
$\Omega \leftarrow_{F_2 \circ F_1} \Omega_2$

(1)1. $\forall f \in (F_2 \circ F_1) \cdot \forall \phi \in \Omega \cdot \exists \phi_2 \in \Omega_2 \cdot \phi \leftarrow f \phi_2$
(2)1. $\exists \phi_2 \in \Omega_2 \cdot \phi \leftarrow (F_2 \circ F_1)$ and $\phi \in \Omega$
(3)1. Choose $f_1 \in F_1$ and $f_2 \in F_2$ such that $f = f_2 \circ f_1$
    Proof: By definition of $\circ$ (Eq. 5).
(3)2. Choose $\phi_1 \in \Omega_1$ such that $\phi \leftarrow f_1 \phi_1$
    Proof: By (3)1 and assumption 1.
(3)3. Choose $\phi_2 \in \Omega_2$ such that $\phi_2 \leftarrow f_2 \phi_2$
    Proof: By (3)1 and assumption 2.
(3)4. $\phi \leftarrow_{F_2 \circ F_1} \phi_2$
    Proof: By (3)2, (3)3, and Lemma 3.1.
(3)5. Q.E.D.
    Proof: By (3)4.
(2)2. Q.E.D.
A PROOFS

Proof: By ∀-rule.

\( \langle 1 \rangle \) 2. Q.E.D.

Proof: By definition of \( \Leftarrow \) (Def. 8).

A.4.1 Lemma 3.1

\( \Leftarrow \) (Def. 7) is transitive.

Assume:
1. \( \phi \Leftarrow f, \phi_1 \)
2. \( \phi_1 \Leftarrow f_2, \phi_2 \)

Prove:
\( \phi \Leftarrow f_2 \circ f_1, \phi_2 \)

Let:
\( f \triangleq f_2 \circ f_1 \)

\( \langle 1 \rangle \) 1. \( \phi_2 \subseteq \{ f(t) \mid t \in \phi \} \)
\( \langle 2 \rangle \) 1. \( \forall t_2 \in \phi_2 \cdot \exists t \in \phi \cdot f(t) = t_2 \)
\( \langle 3 \rangle \) 1. \( \exists t \in \phi \cdot f(t) = t_2 \) for arbitrary \( t_2 \in \phi_2 \)
\( \langle 4 \rangle \) 1. Choose \( t_1 \in \phi_1 \) such that \( f_2(t_1) = t_2 \)

Proof: By assumption 2 and definition of \( \Leftarrow \) (Def. 7).

\( \langle 4 \rangle \) 2. Choose \( t \in \phi \) such that \( f_1(t) = t_1 \)

Proof: By assumption 1 and definition of \( \Leftarrow \) (Def. 7).

\( \langle 4 \rangle \) 3. \( f(t) = t_2 \)

Proof: By transitivity of \( \circ \).

\( \langle 4 \rangle \) 4. Q.E.D.

Proof: By \( \langle 4 \rangle \) 3

\( \langle 3 \rangle \) 2. Q.E.D.

Proof: By ∀-rule.

\( \langle 2 \rangle \) 2. Q.E.D.

Proof: By \( \langle 2 \rangle \) 1.

\( \langle 1 \rangle \) 2. Q.E.D.

Proof: By definition of \( \Leftarrow \) (Def. 7).

A.5 Theorem 2

If there is a valid secure interpretation of a given transformation, then that transformation maintains security.

Assume:
1. \( F \in \mathcal{F} \)
2. \( \Omega_a \Rightarrow_F \Omega_c \)
3. \( R_c(\hat{\Omega}_c, t_c) \Rightarrow \exists t' \in F^{-1}(t_c) : R_a(\hat{\Omega}_a, t') \) for all \( t_c \in \hat{\Omega}_c \)
4. \( (R_c(\hat{\Omega}_c, t_c) \land \forall t' \in \phi_a \cdot t_a \sim \phi \land Q_a(t_a, t')) \Rightarrow \exists \phi_c \in \Omega_c \cdot \forall t' \in \phi_c \cdot t_c \sim t' \land Q_c(t_c, t') \) for all \( t_c \in \Omega_c, \phi \in \Omega_a, \) and \( t_a \in F^{-1}(t_c) \)
5. \( \text{Bsp}_{Q_a, R_c}(\Omega_a) \)

Prove:
\( \text{Bsp}_{Q_a, R_c}(\Omega_a) \)

\( \langle 1 \rangle \) 1. Assume:
1.1 \( t_c \in \hat{\Omega}_c \)
1.2 \( R_c(\hat{\Omega}_c, t_c) \)

Prove:
\( \exists \phi \in \Omega_c \cdot \forall t' \in \phi \cdot t_c \sim \phi \land Q_c(t_c, t') \)

\( \langle 2 \rangle \) 1. Choose \( t_a \in F^{-1}(t_c) \) such that \( R_a(\hat{\Omega}_a, t_a) \)

Proof: By assumptions 1.1, 1.2, and 3.
⟨2⟩2. Choose \( \phi_a \in \Omega_a \) such that \( \forall t' \in \phi_a \cdot t_a \sim_\downarrow t' \land Q(t_a, t') \)

**Proof:** By ⟨2⟩1, assumption 5, and definition of \( \text{Bsp} \) (Def. 6).

⟨2⟩3. Q.E.D.

**Proof:** By assumption 1.2, ⟨2⟩2, and assumption 4.

⟨1⟩2. Q.E.D.

**Proof:** By definition of \( \text{Bsp}\)'s (Def.6).
Chapter 11

Information Flow Security, Abstraction, and Composition

Fredrik Seehusen\textsuperscript{1,2} and Ketil Stølen\textsuperscript{1,2}
\textsuperscript{1} SINTEF ICT, Norway
\textsuperscript{2} University of Oslo, Norway
\{fse, kst\}@sintef.no

December 9, 2008

Abstract

We present a framework that supports an incremental and modular development process of secure software systems. The framework unifies the treatment of secure information flow properties and their relationship to refinement of underspecification, translation from one level of granularity to another, and composition.

1 Introduction

We examine the relationship of information flow security, abstraction, and composition. Information flow security properties are requirements on the flow of information between different security domains (see e.g. [2, 4, 15, 18, 21, 22, 29, 31]). The underlying idea is that an observer residing in one security domain (call it low) shall not, based on its observations, be able to deduce whether behavior associated with another security domain (call it high) has, or has not occurred.

Abstraction and composition are two of the most important notions of software engineering. Abstraction is a means of suppressing irrelevant details. There are two orthogonal kinds of abstraction: abstraction by translation and abstraction by underspecification. Underspecification arises when a term means more than one thing and all these “things” are valid interpretations of that term. For instance, the sentence “a red car” is more concrete than the sentence “a car” because everything that is meant by the former sentence is also meant by the latter\textsuperscript{1}. Translation is a notion that relates specifications described in one vocabulary (call it abstract) to specifications described in another vocabulary (call it concrete) that more closely coincides with the real things one wants to describe. Composition supports the notion separation of concerns in which a (composite) system specification is developed, not as a monolithic entity, but by putting together, or composing, other (basic) specifications.

The question we ask is: how can we preserve secure information flow properties under refinement of underspecification, translation, and composition?

\textsuperscript{1}Underspecification is related to, but different than ambiguity. Ambiguity arises when a term can mean either one thing or another, but not both.
This question is of interest because it is in general desirable to analyze abstract specifications (or basic specifications) w.r.t security as opposed to concrete specifications (or composite specifications). There are three main reasons for this: (1) Analysis is more feasible at the abstract level since the concrete level may include too much detail to make analysis practical. (2) Abstract specifications tend to be more understandable or manageable than concrete specifications, hence it is easier to specify and check security requirements at the abstract levels as opposed to the concrete levels. (3) Abstract specifications are more platform independent than concrete specifications. This means that analysis results are more reusable at the abstract levels.

Items (1) and (2) are applicable for composition as well. This is because composition may result in an explosion of complexity, and because the notion of composition makes system development more manageable.

In answer to the question stated above, we present a framework for specifying secure information flow properties that are preserved under refinement of underspecification. The framework combines the modular structure of Mantel’s framework [15] with the simplicity of the generalized unwinding theorem [2]. In addition we define the notions of translation and composition in a manner that is sufficiently general to capture most translations and composition operators that can be described in terms of trace-semantics, and propose theorems in answer to the question stated above.

The only work that we are aware of that unifies the treatment of information flow security and the notions of refinement of underspecification, translation, and composition is the work of Santen et.al. [8, 26, 24, 25]. However, they address probabilistic security (as opposed to possibilistic security as we do) and their notion of data refinement is less general than our notion of translation.

Apart from the work of Santen et. al. the only other work that we are aware of that considers secure information flow in relation to notions similar to translation is [5, 10]. However, Graham-Cumming and Sanders [5] considers a less general notion of security than we do, and Hutter [10] does not address refinement of underspecification.

Information flow security and refinement of underspecification has been extensively studied in the past. In 1989 it was shown by Jacob [11] that secure information flow properties in general are not preserved by the standard notion of refinement. It has later been observed that the problem originates in the inability of most specification languages to distinguish between underspecification and unpredictability\footnote{Also termed probabilistic nondeterminism [22].} [8, 12, 22]. We show how secure information flow properties in general are preserved under a notion of refinement that takes the distinction of underspecification and unpredictability into consideration. A similar approach is taken in [13, 14], but we consider a more general notion of refinement.

Secure information flow and composition has been previously addressed in [2, 9, 19, 20, 30], but of these papers, only Bossi et. al. [2] considers the problem in light of a general framework of security and a general notion of composition. One of the main differences between our work and the paper by Bossi et. al. is that we consider composition in a semantic model in which the distinction of underspecification and unpredictability is made. This distinction is not made by Bossi et. al., and therefore they cannot rely on this distinction in order to
show that security is preserved by refinement of underspecification. This paper builds on previous work by the authors [28]. However, composition was not considered in [28] and the security framework we used was based on [15]. The framework proposed in this paper is simpler and easier to work with. In summary, the main contributions of this paper are (1) a framework for specifying security information flow properties that are preserved under refinement of underspecification, (2) a characterization of translations that preserve security properties of our framework, and (3) a definition a general notion of composition and conditions under which security properties of our framework are preserved under composition.

This paper is structured as follows: In Sect. 2, we formalize a notion of systems, specifications, and refinement. In Sect. 3, we propose a framework for specifying secure information flow properties for both systems and specifications and show that the framework is sufficiently general to capture many security properties of the literature. Sections 4 and 5 formalize a notion of translation and composition and present conditions under which these notions preserve information flow properties of our framework. In Sect. 6, we discuss related work and in Sect. 7, we provide conclusions and directions of future work. Proofs of all theorems in the paper are given in the appendix.

2 Systems and Specifications

We model the input-output behavior of systems by sequences of events called traces, where an event represents the transmission or the reception of a message. Formally, an event is a pair $(k, m)$ consisting of a kind $k$ and a message $m$. An event whose kind equals $!$ represents the transmission of a message, whereas an event whose kind equals $?$ represents the reception of a message. A message is a triple $(ll_1, ll_2, si)$ consisting of a transmitter $ll_1$, a receiver $ll_2$, and a signal $si$ representing the message body. Both transmitters and receivers are referred to as lifelines, i.e., system entities such as objects or components.

The set of all events is denoted $E$, and the set of all traces is denoted by $T$. We require that the traces of $T$ are causal in the sense that messages must be transmitted before they are received. Throughout this paper, we let $e$ range over $E$ and $s$, $t$, and $u$ range over $T$. A sequence of events, i.e., a trace, is written $(e_1, e_2, ..., e_n)$. The empty trace, i.e., the trace with no events is written $\langle \rangle$.

**Definition 1 (System)** The semantics of a system, denoted $\Phi$, is a set of traces.

In the sequel, we will for short write “system” instead “the semantics of a system” when it clear from the context what is meant.

A specification may describe possible traces that are equivalent in the sense that it is sufficient for a system to fulfill only one of them to be in compliance with the specification. We say that such traces provide potential choices or underspecification. We distinguish such choices from choices that provide unpredictability and that are explicit in sense that they must be fulfilled by a compliant system.

To distinguish between potential and explicit choices of a specification, we interpret a specification, not a trace set, but as a set of trace sets called obligations.
Definition 2 (Specification) A specification, denoted $\Omega$, is a set of trace sets. Each trace set in a specification is called an obligation.

Intuitively, the traces within the same obligation may provide underspecification, while the obligations provide unpredictability in the sense that a compliant system is required to fulfill at least one trace in each obligation of a specification.

Definition 3 (Compliance) A system $\Phi$ is in compliance with specification $\Omega$, written $\Omega \rightsquigarrow \Phi$, iff

$$(t \in \Phi \implies \exists \phi \in \Omega : t \in \phi) \land (\phi \in \Omega \implies \exists t \in \Phi : t \in \phi)$$

A specification $\Omega'$ is a refinement of a specification $\Omega$ if all systems that are in compliance with $\Omega'$ are also in compliance with $\Omega$. In this sense, $\Omega'$ describes its compliant systems more accurately than $\Omega$, ergo it is at least as concrete as $\Omega$.

Definition 4 (Refinement) Specification $\Omega'$ is a refinement of specification $\Omega$, written $\Omega \rightsquigarrow \Omega'$, iff

$$(\forall \phi \in \Omega : \exists \phi' \in \Omega' : \phi' \subseteq \phi) \land (\forall \phi' \in \Omega' : \exists \phi \in \Omega : \phi' \subseteq \phi)$$

This corresponds to so-called limited refinement in STAIRS [23]. For an arbitrary obligation $\phi$ at the abstract level, there must be an obligation $\phi'$ at the concrete level such that $\phi'$ is a refinement of $\phi$ in the sense of the classic notion of refinement (see e.g., [11]). Moreover, each obligation at the concrete level must be a refinement of an obligation at the abstract level. The latter ensures that behavior that was not considered at the abstract level is not introduced at the concrete level.

The following theorem shows how the notions of refinement and compliance are related.

Theorem 1 If specification $\Omega'$ is a refinement of specification $\Omega$ and system $\Phi$ is in compliance with $\Omega'$, then $\Phi$ is in compliance with $\Omega$. Formally,

$$(\Omega \rightsquigarrow \Omega' \land \Omega' \rightsquigarrow \Phi) \implies \Omega \rightsquigarrow \Phi$$

2.1 Example of a specification

Our semantic model for specifications is based on STAIRS [6]. STAIRS provides a formal semantics for UML sequence diagrams. In this section, we explain how a UML sequence diagram can be interpreted as a set of obligations.

Fig. 1 shows an example of a UML sequence diagram. Sequence diagrams describe communication between system entities which we refer to as lifelines. In a diagram, lifelines are represented by vertical dashed lines. An arrow between two lifelines represents a signal being sent from one lifeline to the other in the direction of the arrow. Communication is asynchronous. Therefore, each arrow describes two events: one output event and one input event.

The diagram of Fig. 1 describes two alternative communication scenarios (these are the operands of the $\text{xalt}$ construct). In the first scenario, $\text{UserL}$ stores a document on $\text{Server}$ by sending the signal $\text{storeDocument}$ to it. $\text{Server}$ responds by sending an $\text{ok}$ signal back to $\text{UserL}$. In the second scenario, $\text{UserL}$
and UserH stores and retrieves a document in parallel, respectively. Since the same document cannot be stored on the server while it is being retrieved, it may be the case that a user receives an error signal instead of an ok signal.

In STAIRS, explicit choices are specified by the xalt-construct, whereas potential choices are specified by the alt-construct. This means that a system that is in compliance with the specification of Fig. 1 is required to fulfill both alternatives of the xalt-operator. However, a system that does not produce an error signal is in compliance with the specification, since the error signal is in a potential choice operand.

Semantically, the xalt-operator creates new obligations, whereas the alt operator collapses obligations. For instance, the semantics of the specification that describes lifeline UserL of Fig. 1, written [UserL], is given by

\[
[UserL] = \{\{!s, ?o\}, \{!s, ?o, !s, ?e\}\}
\]

The two obligations correspond to the operands of the xalt construct of Fig. 1. Note that the last obligation contains two traces that provide potential alternatives due to the alt construct. For short, the signal names of Fig. 1 are referred to by their first letter, and the transmitter and the receiver of messages are suppressed. For instance, !s is short for the event (!, (UserL, Server, storeDocument)).

The semantics of the specification DS of Fig. 1 is given by the parallel composition of the specifications that describe each of its three lifelines, i.e.,

\[
[DS] = [UserL] \parallel [Server] \parallel [UserH]
\]

where \parallel denotes the parallel composition operator.

As described previously, traces that provide potential choices may be removed during refinement. For instance, the specification DS' of Fig. 2 is a refinement of specification DS of Fig. 1. In DS', the alternatives in which the server sends out an error message have been removed.
3 INFORMATION FLOW SECURITY

It is possible to refine specification $DS'$ further due to the way in which the parallel operator in defined in STAIRS. That is, the scenario in which $UserL$ stores a document on the server and the scenario in which $UserH$ retrieves a document from the server, may come in any order since the two scenarios are encapsulated by the $par$-operator. In STAIRS, the order in which these scenarios occur are interpreted as potential nondeterministic choices.

The semantics of specification $DS'$ of Fig. 2 is given by

$$
[[UserL] \parallel [Server] \parallel [UserH]] =
\{[s, ?o]\} \parallel \{[s, lo, ?r, ld, ?s, ?r, lo, !d, ?s, ?r, lo, !d, ?s, ?r, lo, !d, ?s, ?r, lo, !d, ?s, ?r, lo, !d, \ldots]\}
$$

Note that the second obligation contains the well-formed traces obtained by interleaving the traces of the operands of the $par$ operator of Fig. 2.

3 Information Flow Security

Information flow security properties (in the following referred to as security properties for short) are requirements on allowed flow of information between different security domains. The underlying requirement is that an observer residing in one security domain (call it low) shall not, based on its observations, be able to deduce whether behavior associated with another security domain (call it high) has, or has not occurred.

In the following, we first (in Sect. 3.1) present a framework for defining security properties for systems. Then, (in Sect. 3.2) we show how standard security properties of the literature can be defined in the framework. Finally, (in Sect. 3.3) we generalize the security framework to specifications and show that all security properties defined in the framework are preserved under refinement.
3 A security framework for systems

In this section, we define a security framework for specifying security properties for systems. First, we present an example which illustrates the notions that underlie information flow security properties.

Example Suppose we require that UserL of Fig. 1 must not deduce that UserH has done something. Thus we let UserL be in the low level domain, and UserH be in the high level domain. Suppose further that UserL can observe its own communication with the server. He can make two low level observations: \( \langle !s, ?o \rangle \) and \( \langle !s, ?e \rangle \). If UserL observes \( \langle !s, ?o \rangle \), he cannot be sure that user UserH has done something even if UserL has complete knowledge of the specification DS of Fig. 1. This is because UserL cannot exclude the fact that the topmost xalt-alternative of Fig. 1 (where UserH is inactive) has occurred when observation \( \langle !s, ?o \rangle \) is made. However, if UserL observes \( \langle !s, ?e \rangle \), he will know for sure that UserH has done something; only the lowermost xalt-alternative of Fig. 1 is compatible with observation \( \langle !s, ?e \rangle \), and UserH is never inactive in this alternative. The specification DS is therefore not secure w.r.t. our requirement. However, the refinement DS' (Fig. 2) of DS, is secure w.r.t. our requirement. In this specification, only one observation can be made (\( \langle !s, ?o \rangle \)), and, as explained above, UserL cannot assert that UserH has done something when observation \( \langle !s, ?o \rangle \) is made. △

The previous example suggests that we need two ingredients to define a security property:

- a notion of low level compatibility;
- a notion of high level behavior.

The notion of low level compatibility is formalized by an equivalence relation \( l \subseteq T \times T \) on traces, and we write \( s \sim_l t \) (i.e., \( (s, t) \in l \)) if the low level observation of \( s \) is compatible with the low level observation of \( t \). The low level equivalence class \( s^l \) of trace \( s \) is the set of all traces that are low level compatible with \( s \), i.e., \( s^l = \{ t \mid s \sim_l t \} \).

High level behavior is similarly defined in terms of an equivalence relation \( h \subseteq T \times T \) on traces, and we write \( s \sim_h t \) if \( s \) and \( t \) are high level compatible, i.e., in the same high level equivalence class. Conversely, we write \( s \not\sim_h t \) (i.e., \( (s, t) \notin h \)) if the traces \( s \) and \( t \) are high level incompatible, i.e., not members of the same high level equivalence class.

To prevent a low level user of a system \( \Phi \) from deducing that high level behavior has occurred, there must for each trace \( t \) and \( s \) of \( \Phi \), be a trace \( u \) in \( \Phi \) such that \( u \) is both low level compatible with \( t \) and high level incompatible with \( s \), i.e.,

\[
\forall s, t \in \Phi : \exists u \in \Phi : s \not\sim_h u \sim_l t
\]  

The presence of \( u \) ensures that the low level user cannot know for certain that the high level behavior class of \( s \) has occurred when observing \( t \). Since \( u \) is low level compatible with \( t \), the low level user cannot know whether \( t \) or \( u \) has occurred when observing \( t \). Since, in addition, \( u \) is high level incompatible with \( s \), the low level user cannot know for sure if the high level class of \( s \) has occurred when observing \( t \). This is illustrated on the left hand side of Fig. 3.
Example In the previous example, we required that UserL must not deduce that UserH has done something. This requirement can be captured by condition (2) for appropriate definitions of l and h.

We define the low level compatibility relation in terms of a set L of low level events that can be observed by the low level user. In the example, UserL can observe its own communication with the server. Therefore, we define L to be set of all events that can occur on the lifeline UserL, i.e., $L = \{!s, ?o, ?e\}$. Low level compatibility is then defined such that two trace are low level compatible if they contain the same low level observations:

$$s \sim_L t \iff s|_L = t|_L$$  \hspace{1cm} (3)

where $s|_L$, the projection of $s$ to $L$, yields the trace obtained from $s$ by removing all events that are not in the set of events $L$.

Since UserL must not deduce that UserH has done something, the high level equivalence class of interest should characterize all traces that contain an event occurring on the lifeline UserH. We let $H$ denote all events that can occur on the lifeline UserH, i.e., $H = \{!r, ?d, ?e\}$, and define the high level equivalence relation $h$ by

$$s \sim_H t \iff s|_H \neq \langle \rangle \land t|_H \neq \langle \rangle$$  \hspace{1cm} (4)

To obtain the high level incompatibility relation, we simply take the complement of $s \sim_H t$, i.e.,

$$s \not\sim_H t \iff s|_H = \langle \rangle \lor t|_H = \langle \rangle$$  \hspace{1cm} (5)

In the previous example, we claimed that specification $DS$ of Fig. 1 was not secure w.r.t. our requirement. We now explain the reason for this by showing that a system $\Phi_{DS}$ that is compliant with $DS$ does not satisfy condition (2).

Let $\Phi_{DS}$ be defined by

$$\Phi_{DS} \triangleq \{(!s, ?s, !o, ?o), (!s, ?s, !o, !r, ?r, !d, ?d), (!r, !r, !s, !s, !e, !e, !d, ?d)\}$$  \hspace{1cm} (6)

Choose traces $s$ and $t$ in $\Phi_{DS}$ such that $s = t = \langle !r, ?r, !s, !s, !e, !e, !d, ?d \rangle$. By condition (2), there must be trace $u$ in $\Phi_{DS}$ such that $u$ is low level compatible with $t$ and high level incompatible with $s$. However, no such $u$ exists. The only trace in $\Phi_{DS}$ that is low level compatible with $t$ is $t$ itself, i.e., $t \cap \Phi_{DS} = \{t\}$, and $t$ is not high level incompatible with $s$. Therefore system $\Phi_{DS}$ is not secure.

In the previous example, we also claimed that the specification $DS'$ of Fig. 2 was secure w.r.t. our requirement. To see why a system $\Phi_{DS'}$ which is compliant with $DS'$ is secure w.r.t. our requirement, define $\Phi_{DS'}$ by

$$\Phi_{DS'} \triangleq \{(!s, ?s, !o, ?o), (!s, ?s, !o, !r, ?r, !d, ?d)\}$$  \hspace{1cm} (7)
Regardless of how we chose $s$ and $t$ in $\Phi_{DS'}$, there is always a trace $u$ in $\Phi_{DS'}$ which is low level compatible with $t$ and high level incompatible with $s$. Therefore system $\Phi_{DS'}$ is secure w.r.t. our requirement. △

As indicated in the beginning of this section, information also flows from the high level domain to the low level domain if the low level observer deduces that a high level behavior has not occurred. To prevent this kind of information flow for a system $\Phi$, there must for each trace $t$ and $s$ of $\Phi$, be a trace $u$ in $\Phi$ such that $u$ is both low level compatible with $t$ and high level compatible with $s$, i.e.,

$$\forall s, t \in \Phi : \exists u \in \Phi : s \sim_h u \sim_l t$$

The presence of $u$ ensures that the low level user cannot deduce that the high level behavior class of $s$ has not occurred. This is pictured on the right hand side of Fig. 3.

Example Let system $\Phi_{DS}$ be given by (6) of the previous example. Suppose that we want to prevent UserL from deducing that UserH has not done something. This requirement is formalized by condition (8) by letting $l$ and $h$ be the low level and high level equivalence relations defined by (3) and (4) in the previous example, respectively. The system $\Phi_{DS}$ is secure w.r.t. this requirement; regardless of the choice of $s$ and $t$ in $\Phi_{DS}$, there is always a trace $u$ in $\Phi_{DS}$ that is low level equivalent to $t$ and high level compatible with $s$. △

Both condition (2) and condition (8) may be seen as schemas that are parameterized by $h$ and $l$ for defining security properties. The conditions are related in the sense that a system $\Phi$ satisfies condition (8) for some $h$ and $l$ if and only if $\Phi$ satisfies condition (2) for the complement $\overline{h}$ (i.e., $\overline{h} = \{(s, t) \mid (s, t) \notin h\}$) and $l$. This means that condition (8) can be expressed in terms of condition (2) and vice versa. It is convenient to work with one condition instead of two. Therefore, we henceforth only work with conditions of the form (8). We now let $h$ denote either an equivalence relation, or the complement of an equivalence relation and write $s \overset{h}{\rightarrow} t$ instead of $s \sim_h t$ or $s \not\sim_h t$ to highlight this.

Many information flow properties of the literature may be expressed as conjunctions of instantiations of condition (8), but not all. The reason for this is that the condition $s \overset{h}{\rightarrow} u \sim_l t$ should sometimes only hold for some traces $s$ and $t$, but not for all. We therefore generalize the condition by introducing a new relation $r$, the restriction relation, and call the new condition a basic security predicate schema.

Definition 5 (Basic security predicate schema for systems) The basic security predicate $Bsp_{rh}(\Phi)$ for restriction relation $r$, high level relation $h$, and low level equivalence relation $l$ holds iff

$$\forall s, t \in \Phi : s \overset{h}{\rightarrow} t \implies \exists u \in \Phi : s \overset{h}{\rightarrow} u \sim_l t$$

The name basic security predicate is taken from [15] where it is shown that many security properties of the literature can be expresses as conjunctions of basic security predicates.

Definition 6 (Security property) A security property $Sp$ is a conjunction of basic security predicates.
Table 1: Basic security predicates of Mantel’s schema

<table>
<thead>
<tr>
<th>Name</th>
<th>$R$</th>
<th>$Q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Re’</td>
<td>true</td>
<td>$u</td>
</tr>
<tr>
<td>Ri’</td>
<td>true</td>
<td>$u</td>
</tr>
<tr>
<td>Ihai’</td>
<td>$e \in \mathcal{HI} \land t = \beta - \alpha \land \alpha</td>
<td>_{\mathcal{H}} = \emptyset \land \gamma’ - \langle e \rangle \in \Phi \land \gamma’</td>
</tr>
<tr>
<td>LAE’</td>
<td>$e \in \mathcal{H} \land t = \beta - \alpha \land \alpha</td>
<td>_{\mathcal{H}} = \emptyset \land \gamma’ - \langle e \rangle \in \Phi$</td>
</tr>
</tbody>
</table>

* The traces $\alpha’’, \beta’’, \gamma’$ are existentially quantified over $T$

### 3.2 Security properties for systems

In this section, we define the security properties non-inference $Nf$ [21], generalized non-inference $Gnf$ [20], generalized non-interference $Gni$ [18], and the perfect security property $Psp$ [31] in our framework.

The idea of defining security properties as a conjunction of basic security predicates was introduced by Mantel in [15]. For this reason, we will first present the above mentioned security properties as they are defined by Mantel’s framework. Then we present the corresponding definitions in our framework.

In Mantel’s framework, a basic security predicate is given by the following definition.

**Definition 7 (Mantel’s basic security predicate schema)** System $\Phi$ satisfies the basic security predicate $Bsp’_{RQ}(\Phi)$ for restriction $R$ and closure requirement $Q$ iff

$$\forall t \in \Phi, \alpha, \beta \in T, e \in \mathcal{E} : (R(\Phi, t, \alpha, \beta, e) \implies \exists u \in LLES(\Phi, t) : Q(t, u, \alpha, \beta, e))$$

where $LLES$, the so-called low level equivalence set, is defined by

$$LLES(\Phi, t) \triangleq t|_L \cap \Phi$$

We make some initial assumptions before we present the basic security predicates of Mantel’s schema that are needed to define the properties $Nf$, $Gnf$, $Gni$, and $Psp$. First, we assume that there is a set of low level events $L$ that can be observed by the low level user, and a set $H$ disjoint from $L$ of high level events that are to be considered as confidential. The subset $\mathcal{HI}$ of $H$ will denote the set of all high level input events. For simplicity, we shall assume that the set of all events $\mathcal{E}$ equals $L \cup H$.

Throughout this section we assume a fixed low level equivalence relation defined

$$s \sim_t t \iff s|_L = t|_L \quad (9)$$

and we assume that all systems $\Phi$ are prefix-closed$^3$ (this assumption is made in [15].)

Table 1 shows the four basic security predicates of Mantel’s schema that are needed to define the security properties. These are Re’ (Removal of Events), Ri’ (Removal of Inputs), Ihai’ (Insertion of High level Admissible Inputs), and

$^3$ $\Phi$ is prefix closed if $t \in \Phi \land s \preceq t \implies s \in \Phi$. 
Table 2: Basic security predicates of our schema

<table>
<thead>
<tr>
<th>Name</th>
<th>$\tau$</th>
<th>$\mathfrak{h}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Re}$</td>
<td>true</td>
<td>$s</td>
</tr>
<tr>
<td>$\text{Ri}$</td>
<td>true</td>
<td>$s</td>
</tr>
<tr>
<td>$\text{Ihai}$</td>
<td>$t = \beta \cdot \alpha \land \alpha</td>
<td><em>{\mathcal{H}} = (\bot) \land s = \gamma' = \langle e' \rangle \land \gamma'</em>{</td>
</tr>
<tr>
<td>$\text{Iae}$</td>
<td>$t = \beta \cdot \alpha \land \alpha</td>
<td>_{\mathcal{H}} = (\bot) \land s = \beta \cdot \langle e \rangle$</td>
</tr>
</tbody>
</table>

* The traces $\alpha, \beta$ and $\gamma'$ is existentially quantified over $T$, $e'$ is existentially quantified over $H$, $e$ is existentially quantified over $\mathcal{H}$, and $\alpha'$ is existentially quantified over $L^*$.

IAE’ (Insertion of Admissible Events). See [15] for an explanation of these. In Table 1, Name refers to the name of the basic security predicate and $R$ and $Q$ refer to the restriction and the closure requirement that are used to instantiate the schema. Variables that are not quantified in the table refer to the variables of Def. 5. For instance, the definition of the basic security predicate $\text{Re}'$ is

$$\text{Re}'(\Phi) \triangleq \forall t \in \Phi, \alpha, \beta \in T, e \in E : \text{true} \implies \exists u \in LLES(\Phi, t) : u|_{\mathcal{H}} = (\bot)$$

Due to [15], we know that the following Theorem holds.

**Theorem 2 ([15])** The following equivalences hold:

- $\text{GNF}(\Phi) \iff \text{Ri}'(\Phi)$
- $\text{NF}(\Phi) \iff \text{Re}'(\Phi)$
- $\text{GNI}(\Phi) \iff (\text{Ri}'(\Phi) \land \text{Ihai}'(\Phi))$
- $\text{PSP}(\Phi) \iff (\text{Re}'(\Phi) \land \text{IAE}'(\Phi))$

The NF property ensures that the low level user cannot deduce that a high level event has occurred. The GNF property is similar except that it treats high level input events as confidential as opposed to all high level events. As explained in [15], GNI demands that any interleaving of the high level input of one trace with the low level behavior of another trace can be made a possible trace by adapting the outputs. The PSP property ensures the low level user cannot deduce that a high level event has occurred (the RE part) or that any high level event which is not influenced by the low level user has not occurred (the IAE part).

Table 2 shows the corresponding basic security predicates of our schema. Variables that are not quantified in the table refer to the variables of Def. 5. For instance, the definition of the basic security predicate $\text{Re}$ is

$$\text{Re}(\Phi) \triangleq \forall s, t \in \Phi : \text{true} \implies \exists u \in \Phi : (s|_{\mathcal{H}} = (\bot) \lor u|_{\mathcal{H}} = (\bot)) \land u \sim_{t} t$$

In Table 2, the high level relation $\mathfrak{h}$ for RE and $\text{Re}'$ may be seen as the complement of the equivalence relation defining the set of all traces that contain a high level event or a high level input event, respectively. Note that the definition of $\mathfrak{h}$ in IAE makes use of the function $h \in T \rightarrow T$ which yields the high level behavior of a given trace. More precisely, $h(t)$ yields the trace obtained from $t$ by replacing all low level events by the dummy event $\sqrt{1}$, and removing all dummy-events occurring after the last high level event in $t$. If $t$ does not
contain any high level events, then \( h(t) = \langle \rangle \). For example, for \( e_l, e'_l \in \mathcal{L} \), and \( e_h, e'_h \in \mathcal{H} \), we have
\[
h((e_l, e_h, e'_l, e_l, e_h)) = (\sqrt{e_h}, e'_h, \sqrt{e_h})
\]

Intuitively, \( h(s) = h(u) \) holds when \( s \) and \( t \) contain the same timing sensitive sequences of high level events.

**Theorem 3** The following equivalences hold:
\[
\begin{align*}
\text{Re}(\Phi) & \iff \text{Re}'(\Phi) \\
\text{Ri}(\Phi) & \iff \text{Ri}'(\Phi) \\
\text{Ihai}(\Phi) & \iff \text{Ihai}'(\Phi) \\
\text{IAE}(\Phi) & \iff \text{IAE}'(\Phi)
\end{align*}
\]

### 3.3 A security framework for specifications

In this section, we define a security framework for specifications. This allows us to exploit the distinction of potential and explicit choices to define security properties that are preserved under refinement. Closing security properties under refinement without making this distinction would make the properties too strong to be useful. To see this, consider the the standard notion of refinement by underspecification which has previously been used in investigating the relationship of information flow security and refinement (see e.g., [11]). It states that a set of traces \( \Phi' \) is a refinement of a trace set \( \Phi \) iff
\[
\Phi' \subseteq \Phi
\]

The reason why secure information flow properties are not preserved by this notion refinement of underspecification becomes apparent when one considers again the manner in which these properties are defined (see Def. 5); \( \Phi \) is secure if for each of its traces \( s \) and \( t \) satisfying the restriction \( r \), there is a trace \( u \) in \( \Phi \) whose presence ensures that \( s \) and \( t \) do not compromise security. However, by (10) there is no guarantee that a refinement of \( \Phi \) will include those traces that ensure the security of \( \Phi \), hence secure information flow properties are in general not preserved by refinement.

Intuitively, the cause of this problem is that security properties depend on unpredictability. For instance, the strength of one’s password may be measured in terms of how hard it is for an attacker to guess the password one has chosen. The demand for the presence of the traces \( u \) may be seen as the security predicate’s requirement of unpredictability, but traces that provide this unpredictability may be removed during refinement.

Clearly, closing security properties under the standard notion of refinement would be make the properties too strong to be useful. This motivates a re-definition of the basic security schema which takes into account the distinction between underspecification and unpredictability.

**Definition 8 (Basic security predicate schema for specifications)** The basic security predicate \( \text{BSP}_{rhl}(\Omega) \) for restriction relation \( r \), high level relation \( h \),
and low level equivalence relation \( \sim \) holds iff
\[
\forall s, t \in \hat{\Omega} : s \overset{\sim}{\rightarrow} t \implies \exists \phi \in \Omega : \{ s \} \overset{\sim}{\rightarrow} \phi \sim \{ t \}
\]

Instead of demanding that there is a trace \( u \) that satisfies some criterion, we demand that there is an obligation \( \phi \) such that all its traces satisfy that criterion. The intuition of (Def. 8) is that obligations, as opposed to individual traces, are understood as providing the unpredictability required by instances of the schema. It follows from this that the following theorem holds.

**Theorem 4** Bsp\(_{rlh} \) is preserved by refinement for arbitrary restriction relation \( r \), high level relation \( h \), and low level equivalence relation \( \sim \):
\[
\Omega \rightarrow \Omega' \land \text{Bsp}\(_{rlh} \)(\Omega) \implies \text{Bsp}\(_{rlh} \)(\Omega')
\]

The basic security predicate schema for specifications is a generalization of the basic security schema for systems in the sense of the following theorem.

**Theorem 5** Let \( \Phi \) be a system, and let \( \Omega \) be the specification obtained from \( \Phi \) by creating a new obligation for each trace in \( \Phi \), i.e., \( \Omega = \{ \{ t \} \mid t \in \Phi \} \). Then \( \Omega \) satisfies the basic security predicate \( \text{Bsp}\(_{rlh} \)(of \text{Def. 5}) \) if and only if \( \Phi \) satisfies \( \text{Bsp}\(_{rlh} \)(of \text{Def. 8}) \), i.e.,
\[
\text{Bsp}\(_{rlh} \)(\Phi) \iff \text{Bsp}\(_{rlh} \)(\Omega)
\]

Since a security property is a conjunction of basic security predicates and all security predicates are preserved under refinement, then clearly all security properties are preserved under refinement as well.

**Corollary 1** All security properties are preserved under refinement.

**Example** In Sect. 3.2, we formalized the requirement that \( UserL \) must not deduce that \( UserH \) has done something by instantiating condition (2) by the equivalence relation \( \sim \) and \( \theta \) defined by (3) and (4), respectively. This condition is in fact the Nf property as defined in Sect. 3.2. The specification \( DS' \) of Fig. 2 is secure w.r.t. the Nf property defined for specifications when the set \( \mathcal{L} \) of low level events is defined as all the events that can occur on the lifeline of \( UserL \) and the set \( \mathcal{H} \) of high level events is defined as all the event that can occur on lifeline \( UserH \). To see this, recall the semantics of specification \( DS' \) of Fig. 2 as given by (1) in Sect. 2.1. The low level user can make the single observation \( \langle l_s, ?o \rangle \). Since \( \llbracket DS' \rrbracket \) contains the obligation \( \{ \langle l_s, ?s, l_o, ?o \rangle \} \) whose trace is both low level compatible with the observation and contains no high level events, \( UserL \) cannot deduce with certainty that \( UserH \) has done something when observing \( \langle l_s, ?o \rangle \). The specification \( DS' \) can be refined further, for instance to ensure that server replies to the users immediately after a request has been received. By Theorem. 4, we know that any such refinement is secure w.r.t. the Nf-property.

We note that it is also the case that the specifications \( \llbracket UserL \rrbracket \), \( \llbracket Server \rrbracket \), and \( \llbracket UserH \rrbracket \) of Fig. 2 are secure w.r.t. the Nf-property. \( \triangle \)

---

4The operator \( \sim \) collapses a set of obligations into a set of traces, i.e. \( \tilde{\Omega} = \bigcup_{\phi \in \Omega} \phi \). Also note that for trace sets \( A \) and \( B \) and relation some relation \( \epsilon \) on traces, we write \( A \epsilon B \) iff \( a \epsilon b \) for all \( a \) in \( A \) and \( b \) in \( B \).
4 Translation

The vocabulary that is used in describing a system is always an abstraction of the real things one wants to describe. The more closely the vocabulary coincides with these things, the more concrete it is. For example, the behavior of a human may be described in terms of a vocabulary consisting sequences of actions such as eat, sleep, walk and so on. The behavior of the same human may also be described by sequences of actions performed by a collection of cells. The notion of translation essentially relates specifications described in one vocabulary to specifications described in another vocabulary of a finer granularity.

We model the abstract vocabulary by a set $T$ of abstract traces, and the concrete vocabulary by a set $T'$ of concrete traces. Attention is henceforth restricted to translation functions $f \in T \rightarrow (\mathcal{P}(T') \setminus \emptyset)$ mapping abstract traces to sets of concrete traces and translations indirectly built from such. A translation function $f$ is lifted to trace sets as follows

$$f(\phi) = \{ t' \in f(t) \mid t \in \phi \}$$

**Definition 9 (Translation)** A translation is a set $F$ of translation functions. The (concrete) specification $F(\Omega)$ obtained by applying $F$ to (abstract) specification $\Omega$ is defined

$$F(\Omega) \triangleq \{ f(\phi) \mid \phi \in \Omega \land f \in F \}$$

We use sets of translation functions on traces to model the fact that one abstract obligation may be translated to several concrete obligations. Technically, this is more convenient than using an obligation set valued function on obligations.

The aim of this section is to characterize translations that allow us to exploit the security of the abstract level in establishing the security at the concrete level. The abstract specification $\Omega$ and its translation $F(\Omega)$ may be in different semantic domains; $F(\Omega)$ may include particularities that are not present in $\Omega$. Consequently, the security requirement at the abstract level may be different than the security requirement at the concrete level since the requirement at the concrete level must take these particularities into account. We therefore distinguish between abstract security requirements and concrete security requirements. More formally, we let the abstract security requirement (denoted $A$) be a basic security predicate that is instantiated by some fixed but arbitrary relations $r$, $h$, and $l$ on $T$. Similarly, we let the concrete security requirement (denoted $C$) be a basic security predicate that is instantiated by some fixed but arbitrary relations $r'$, $h'$, and $l'$ on $T'$. We have

$$A \triangleq \text{Bsp}_{rhl} \quad C \triangleq \text{Bsp}_{r'h'l'}$$

We are interested in conditions on transformations that allow us to exploit the fact that $\Omega$ is secure w.r.t. $A$ in order to show that $F(\Omega)$ is secure w.r.t. to $C$. What does this mean? Consider the diagram below.

```
\begin{align*}
    s \xrightarrow{a} \{s\} & \xrightarrow{b} \phi \sim_l \{t\} \\
    x \quad \downarrow y \\
    s' \xrightarrow{a} \{s'\} & \xrightarrow{b'} \phi' \sim_{l'} \{t'\}
\end{align*}
```

Note that the diagram is informal; it is only meant as an illustration.
By Def. 8, the implication labeled $a$ holds for all $s, t \in \hat{\Omega}$ if we know that $\Omega$ is secure w.r.t. $A$. Showing that $F(\Omega)$ is secure w.r.t. $C$ corresponds to showing that the implication labeled $c$ holds for traces in $F(\Omega)$. Exploiting the fact that security has been established at the abstract level therefore means that one takes advantage of the implication $a$ in order to show $c$. We can do this by showing that the implications labeled $x$ and $y$ hold, because if $x$, $a$, and $y$ hold, then $c$ holds by transitivity. This results in the following theorem.

**Theorem 6** Specification $F(\Omega)$ is secure w.r.t. $Bsp_{r'lh'}$ if $\Omega$ is secure w.r.t. $Bsp_{rlh}$ and $F$ is a translation that satisfies the following conditions for all $f_1, f_2 \in F$, $s, t \in \hat{\Omega}$, $\phi \in \Omega$, $s' \in f_1(s)$, and $t' \in f_2(t)$

$$s' \xrightarrow{\sim} t' \Rightarrow s \xrightarrow{\sim} t$$

$$\{s\} \overset{a}{\xrightarrow{\sim r}} \{t\} \Rightarrow \exists f \in F : \{s'\} \overset{h'}{\xrightarrow{\sim r}} f(\phi) \sim \{t'\}$$

**4.1 Example of translation**

Preservation of security under translation enables security to be established for an abstract specification without having to reestablish security for its concrete translation. This is desirable since analyzing an abstract specification is usually cheaper than analyzing its more detailed translation. Another reason is that abstract specifications are more platform independent than concrete specifications. This means that analysis results can be reused for abstract specifications that are translated to several different platforms. Since all security properties defined in framework are preserved under refinement, our approach offers the benefit that security is preserved under a series of successive translation and refinement steps.

In this section, we give an example of a translation and we exploit Theorem 6 to show that the translation preserves the $\text{Nf}$-property.

Consider again Fig. 2. Imagine that the communication between the users and the server is based on a high level communication protocol. In practice, high level communication protocols are bound to lower level protocols that handle issues such as message disappearance, message overtaking, message fragmentation, and error correction.

The process of binding a high level communication protocol (call it $A$) to a lower-level protocol (call it $C$) can be described by a translation that takes a
transformation at the $A$ level as input and yields a specification at the $C$ level as output.

We give an example of a lower-level protocol $C$ which can be used to handle message disappearance. The $C$ protocol is governed by the following rules: (1) The reception of each message is explicitly acknowledged by the receiver, i.e., the receiver sends an acknowledgement message ($ack$) back to the transmitter to indicate that the message has been received. (2) If the transmitter of a message does not receive an acknowledgement, then the message is retransmitted. (3) The transmitter of a message gives up after transmitting the same message twice.

In Fig. 4, we have illustrated what an output event at the $A$ level (on the left hand side of Fig. 4) and an input event at the $A$ level (on the right hand side of Fig. 4) corresponds to at the $C$ level. Note that the lifeline $x$ in the figure represents an arbitrary lifeline. For output events, there are three alternatives. The first (upper most) alternative describes the case in which a message is transmitted whereupon an acknowledgement ($ack$) is received. In the second alternative, no acknowledgement is received after the message has been transmitted once. The message is therefore retransmitted, whereupon an acknowledgement is received. In the third alternative, the sender transmits the message twice and gives up because no acknowledgement is received. As further illustrated on the right hand side of Fig. 4, input events at the $A$ level correspond to one possible scenario at the $C$ level in which an explicit acknowledgement is sent after a message has been received.

A translation from a specification $\Omega$ at the $A$ level to a specification at the $C$ level replaces all events in $\Omega$ by their corresponding behavior at the $C$ level. In the following we formalize this translation for specifications describing the behavior of a single lifeline. When then show that the translation preserves the Nf-property. Later, in Sect. 5.1, we show how to formalize the translation for specifications consisting of more than one lifeline.

First, we formalize the translation of an output event at the $A$ level to its corresponding behavior at the $C$ level. Because output events at the $C$ level correspond to three different alternatives that are specified as explicit nondeterministic choices (due to the $xalt$-operator), we need three functions, $oe_1$, $oe_2$, and $oe_3$, one function for each alternative. These functions are defined by

\[
\begin{align*}
    oe_1(!m) & \triangleq \{(!m, ?ack)\} \\
    oe_2(!m) & \triangleq \{(!m, !m, ?ack)\} \\
    oe_3(!m) & \triangleq \{(!m, !m)\}
\end{align*}
\]

The translation of input events is defined by the function $ie$ (only one function is needed since an input event only corresponds to one alternative at the $C$ level) as follows

\[
ie(?m) \triangleq \{(?m, !ack)\}
\]

We let $a2c_\rho$ be the translation function that takes a trace at the $A$ level, and applies $ie$ to all input event of the trace, and $oe_1$, $oe_2$, or $oe_3$ to each output event of the trace in the order given by the infinite sequence $\rho$ over the set $\{1, 2, 3\}$ (representing the three alternative output event scenarios at the $C$
level). Formally,
\[
\begin{align*}
    a2c_\rho(\langle \rangle) & \triangleq \{\langle \rangle\} \\
    a2c_\rho(\langle \! (\! m \! ) \! \rangle \! - \! s) & \triangleq \alpha c_i(\langle \! m \! \rangle \! - \! a2c_\rho(s) \\
    a2c_\rho(\langle ?m \rangle \! - \! s) & \triangleq \iota c(\langle ?m \rangle \! - \! a2c_\rho(s)
\end{align*}
\]

The translation of a single lifeline specification at the A level to the C level, is given by the set of translation functions \( A2C \) defined as follows
\[
A2C \triangleq \{ a2c_\rho \mid \rho \in \{1, 2, 3\}^\infty \}
\]
where \( B^\infty \) yields the set of all infinite sequences of elements in the set \( B \).

Let \( \Omega \) denote \([\text{User}L], [\text{Server}], \) or \([\text{User}H] \) of Fig. 2. To show why the translation \( A2C \) preserves the NF-property for a specification \( \Omega \), we instantiate the conditions of Theorem 6 according to the definition of \( NF \). For simplicity, we define \( NF \) as the basic security predicate instantiated by the restriction relation \( r \), high level relation \( h \), and low level relation \( l \) defined by
\[
s \overset{r}{\rightarrow} t \Leftrightarrow \text{TRUE} \quad s \overset{l}{\rightarrow} t \Leftrightarrow t|_H = \langle \rangle \quad s \overset{l}{\sim} t \Leftrightarrow s|_L = t|_L
\]
This definition of \( NF \) is equivalent to the definition given in Sect. 3.2, but the definition of \( h \) given here simplifies the instantiation of the conditions of Theorem 6. To instantiate the conditions, we lift the projection operator to trace sets such that \( \Phi|_E \) yields the set obtained from \( \Phi \) by projecting each trace in \( \Phi \) to \( E \), i.e., \( \Phi|_E = \{ t|_E \mid t \in \Phi \} \). We get
\[
\exists a2c' \in A2C : a2c'(\phi)|_H = \{\langle \rangle\} \land a2c'(\phi)|_L = \{\langle \rangle\} \Rightarrow \exists a2c' \in A2C : a2c'(\phi)|_L = \{\langle \rangle\} \Rightarrow a2c(t)|_L (12)
\]
for all \( t \in \tilde{\Omega} \), \( \phi \in \Omega \), \( a2c \) in \( A2C \), and \( t' \) in \( a2c(t) \).

Note that by definition of the restriction \( r \) for the NF-property, the upper most condition of Theorem 6 is trivially satisfied for all translations. Thus this condition can be ignored.

The first part of condition (12) requires that an obligation at the A level which does not contain any high level events, is not translated to an obligation at the C level that contains a high level event. This condition is satisfied by any \( A2C \) because the set \( H \) of high level events is assumed to be all events that can occur on a single lifeline. Translation \( A2C \) cannot change the lifeline that an event occurs on. It follows from this that condition (12) is reduced to
\[
\phi|_L = \{\langle \rangle\} \Rightarrow \exists a2c' \in A2C : a2c'(\phi)|_L = a2c(t)|_L (13)
\]
for all \( t \in \tilde{\Omega} \), \( \phi \in \Omega \), and \( a2c \) in \( A2C \).

This condition holds because the set \( L \) of low level events is assumed to be all events that can occur on lifeline UserL. Since we only consider translation of single lifeline specifications in this section, this means that we either have \( \phi|_L = \{\langle \rangle\} \) and \( t|_L = \{\langle \rangle\} \) in the case where \( \Omega \) denotes \([\text{Server}]\) or \([\text{User}H] \), or we have \( \phi|_L = \{\langle \rangle\} \Rightarrow \phi = \{\langle \rangle\} \) in the case where \( \Omega \) denotes \([\text{User}L] \).

\(^7\)The operator \( - \) yields the concatenation of two sequences. The operator is lifted to sets of sequences such that \( A - B \) yields the set obtained by concatenating all sequences in \( A \) by all sequences in \( B \).
In either case, it is easy to see that condition (13) holds. Thus we know by Theorem 6 and definition of \( Nf \)-property, that the translation \( A2C \) preserves the \( Nf \)-property.

When the translation \( A2C \) is applied to, say the specification \([\text{Server}]\), all events of the specification are replaced by their corresponding behavior at the \( C \) level. It is still possible to refine \( A2C([\text{Server}]) \) by removing underspecification, for instance to ensure that all reception messages are immediately followed by an acknowledgement. However, this refinement will not render the specification insecure w.r.t. the \( Nf \) property since all refinements are security preserving.

## 5 Composition

In any reasonable specification language, a (composite) specification may be obtained by putting together, or as we shall say, composing, (basic) specifications. Standard composition operators include parallel composition, sequential composition, and nondeterministic choice. However, we do not restrict attention to any particular operator. Composition is instead considered in the abstract, and defined in terms of binary operators \( \odot \in T \times T \rightarrow (P(T) \setminus \emptyset) \) on traces. Any trace operator \( \odot \) is lifted to trace sets as follows

\[
\phi_1 \odot \phi_2 \triangleq \{ s \in (s_1 \odot s_2) | s_1 \in \phi_1 \land s_2 \in \phi_2 \}
\]

**Definition 10 (Composition operator)** A composition operator is a set \( \odot \) of trace operators. The composition of specification \( \Omega_1 \) and specification \( \Omega_2 \), written \( \Omega_1 \odot \Omega_2 \), is defined

\[
\Omega_1 \odot \Omega_2 \triangleq \{ \phi_1 \odot \phi_2 | \phi_1 \in \Omega_1 \land \phi_2 \in \Omega_2 \land \odot \in \odot \}
\]

The definition captures standard binary composition operators.

**Example** The operators which are used in STAIRS for specifying potential nondeterministic choice (\( \text{alt} \)), explicit nondeterministic choice (\( \text{xalt} \)), and parallel composition (\( \text{par} \)) are defined by

\[
\begin{align*}
[d_1 \text{alt} d_2] & \triangleq \{ \phi_1 \cup \phi_2 | \phi_1 \in \llbracket d_1 \rrbracket \land \phi_2 \in \llbracket d_2 \rrbracket 
\llbracket d_1 \text{xalt} d_2 \rrbracket & \triangleq \llbracket d_1 \rrbracket \cup \llbracket d_2 \rrbracket 
\llbracket d_1 \text{par} d_2 \rrbracket & \triangleq \{ \phi_1 \parallel \phi_2 | \phi_1 \in \llbracket d_1 \rrbracket \land \phi_2 \in \llbracket d_2 \rrbracket 
\end{align*}
\]

where \( \phi_1 \parallel \phi_2 \) yields all well-formed interleavings of all traces in \( \phi_1 \) and \( \phi_2 \) (see [7]).

All these definitions can be captured by our notion of composition operator (Def. 10). We define potential nondeterministic choice by operator \( \odot_{\text{alt}} \triangleq \{ \odot_{\text{alt}} \} \) where \( \odot_{\text{alt}} \) is defined by

\[
s \odot_{\text{alt}} t \triangleq \{ s, t \}
\]

Explicit nondeterministic choice is defined by operator \( \odot_{\text{xalt}} \triangleq \{ \odot_{\text{xalt}}_1, \odot_{\text{xalt}}_2 \} \) where

\[
s \odot_{\text{xalt}}_1 t \triangleq \{ s \}, \quad s \odot_{\text{xalt}}_2 t \triangleq \{ t \}
\]

Finally, parallel composition is defined by \( \odot_{\text{par}} \triangleq \{ \odot_{\text{par}} \} \) where \( s \odot_{\text{par}} t \) yields the set of all well-formed interleavings of its arguments (see [7]). \( \triangle \)
We characterize compositions that allow us to exploit the security of the basic specifications in establishing the security of the composite specification. We follow the same line of reasoning as in the previous section. This is because the formal definition of a composite operator is of the same form as the definition of a translation (Def. 9). The only essential difference is that a composition operator takes two arguments whereas a translation takes one argument. Hence, we need only to generalize the results of the previous section such that they apply to operators with two arguments instead of one.

In the following we assume three fixed but arbitrary security requirements, one for each basic specification, and one for the composite specification:

\[ B_1 \triangleq \mathsf{Bsp}_{\tau_1, b_1, t_1}, \quad B_2 \triangleq \mathsf{Bsp}_{\tau_2, b_2, t_2}, \quad C \triangleq \mathsf{Bsp}_{\text{oth}} \]

We want to exploit the fact that the basic specification \( \Omega_1 \) is secure w.r.t. \( B_1 \) and that the basic specification \( \Omega_2 \) is secure w.r.t. \( B_2 \) in order to show that the composition \( \Omega_1 \circ \Omega_2 \) is secure w.r.t. \( C \). This scenario is illustrated in the diagram below.

As in the previous section, the implications labeled \( b_1 \), \( b_2 \), and \( c \) are understood to hold whenever \( B_1(\Omega_1) \), \( B_2(\Omega_2) \), and \( C(\Omega_1 \circ \Omega_2) \) hold, respectively. Thus if the basic specifications \( \Omega_1 \) and \( \Omega_2 \) are secure, then the implications labeled \( b_1 \) and \( b_2 \) hold, and we can take advantage of this in order to establish the implication \( c \). As indicated in the diagram, we can do this by ensuring that the implications labeled \( x_1 \), \( x_2 \), \( y_1 \), and \( y_2 \) hold. The conditions under which the implications hold are essentially the same as in the previous section except for the fact that we have to take two arguments into consideration instead of one.

**Theorem 7** \( \Omega_1 \circ \Omega_2 \) is secure w.r.t. \( B_{\text{oth}} \) if \( \Omega_1 \) is secure w.r.t. \( B_{\text{oth}} \). \( \Omega_2 \) is secure w.r.t. \( B_{\text{oth}} \), and \( \circ \) is a composition operator that satisfies the following conditions for all \( \phi_1, \phi_2 \in \Diamond, s_1, t_1 \in \overline{\Omega}_1, \phi_1 \in \Omega_1, s_2, t_2 \in \Omega_2, \phi_2 \in \Omega_2, s \in (s_1 \circ_1 s_2), t \in (t_1 \circ_2 t_2) \)

\[ s \xrightarrow{\tau} t \implies (s_1 \xrightarrow{\tau_1} t_1 \wedge s_2 \xrightarrow{\tau_2} t_2) \]

\[ (s_1) \xrightarrow{b_1} \phi_1 \sim_{t_1} \{ t_1 \} \wedge (s_2) \xrightarrow{b_2} \phi_2 \sim_{t_2} \{ t_2 \} \implies \exists \diamond \in \Diamond : (s) \xrightarrow{b} (\phi_1 \circ \phi_2) \sim_{t} \{ t \} \]

### 5.1 Example of Composition

Preservation of security under composition enables the security of the specification as a whole to be established by analyzing each part of the specification in...
Again, since \( L \) of the specification.

received in the same order that they are sent is a perfectly valid implementation on the other hand, a system that for instance ensures that messages are always lying communication medium of a specification is underspecified. For instance, whether messages are received in the same order that they are sent. On the one hand, the specification must take all these alternatives into account, but

whether message may disappear during transmission, or nondeterministic alternatives to be created. This may be useful when the underlying communication medium of a specification is underspecified. For instance, one might not know whether message may disappear during transmission, or

whether message may disappear during transmission, or. For instance, the translation of specification \( DS' \) (Fig. 2) to the \( C \) level is defined by

\[
DS' = A2C([UserL]) \odot_{par}(A2C([Server]) \odot_{par} A2C([UserH]))
\]

In order to show that \( DS' \) is secure w.r.t. the \( Nf \)-property, we may check if the parallel composition operator \( \odot_{par} \) preserves the \( Nf \)-property. This is sufficient because we have already shown that the specifications \([UserL],[Server],\) and \([UserH]\) of Fig. 2 are secure w.r.t. the \( Nf \)-property (see Sect. 3.3) and that the translation \( A2C \) preserves the \( Nf \)-property (see Sect. 4.1).

To show that the \( Nf \) property is preserved under the \( \odot_{par} \) operator, we instantiate the conditions of Theorem 7 and get

\[
\phi_{l}|_{H} = \{\} \land \phi_{l}|_{C} = \{t_{i}\}|_{C} \land \phi_{sh}|_{H} = \{\} \land \phi_{sh}|_{C} = \{t_{sh}\}|_{C} \implies \phi_{l} \land \phi_{sh]|_{C} = \{t\}|_{C}
\]

for all \( t_{i} \) and \( \phi_{l} \) in \( A2C([UserL]) \), \( t_{sh} \) and \( \phi_{sh} \) in \( A2C([Server]) \odot_{par} A2C([UserH]) \), and \( t_{i} \) in \( \{t_{i}\} \).

The first part of condition (15) is trivially satisfied because the interleaving of traces in two obligations that do not contain any high level events, will not contain any high level events either. The second part of condition (15) is satisfied because the set \( L \) of low level events is assumed to be all events occurring in \( A2C([UserL]) \). Thus for all obligations \( \phi_{l} \) and traces \( t_{i} \) in \([UserL]\), we know that \( \phi_{l}|_{C} = \{t_{i}\}|_{C} \iff \phi_{l} = \{t_{i}\} \).

It follows that condition (15) is reduced to

\[
\{\{t_{i}\} \odot_{par} \phi_{sh}\}|_{C} = \{\{t_{i}\} \odot_{par} \{t_{sh}\}\}|_{C}
\]

for all \( t_{i} \) and \( \phi_{l} \) in \( A2C([UserL]) \), \( t_{sh} \) and \( \phi_{sh} \) in \( A2C([Server]) \odot_{par} A2C([UserH]) \), and \( t_{i} \) in \( \{t_{i}\} \).

Again, \( L \) is exactly the set of events that can occur in \([UserL]\), we know that \( \phi_{sh}|_{C} = \{\} \) and \( \{t_{sh}\}|_{C} = \{\} \). Thus for each trace \( u' \) in \( \{\{t_{i}\} \odot_{par} \phi_{sh}\} \) or \( \{\{t_{i}\} \odot_{par} \{t_{sh}\}\} \), we know that \( u'|_{C} = t_{i} \), which again means that condition (16) is satisfied.

The parallel composition of two specification may result in new potential nondeterministic alternatives to be created. This may be useful when the underlying communication medium of a specification is underspecified. For instance, isolation. This is usually less expensive than establishing security of the specification as a whole. Our approach offers the advantage that composition may be used together with translation and refinement in such a way that security is preserved. Also, any additional nondeterminism that arises as a result of composition will not violate the security of the specification. This is because all security properties defined in our framework are preserved under refinement.
As an example, the parallel composition of the two specifications \( \{\langle !a, b \rangle \}, \{\langle ?a, ?b \rangle \} \) and \( \{\langle !c, !d \rangle \}, \{\langle ?c, ?d \rangle \} \), yields
\[
\{\langle !a, b, ?a, ?b \rangle, \langle !a, b, ?a, ?b \rangle \},
\{\langle !c, !d, ?c, ?d \rangle, \langle !c, !d, ?c, ?d \rangle \}
\]
Notice that messages are not necessarily received in the same order that they are sent, but that the alternative orders in which messages are received are specified as potential choices. This means the following specification in which messages are received in the same order they are sent is a refinement of the above:
\[
\{\langle !a, b, ?a, ?b \rangle, \langle !a, b, ?a, ?b \rangle \}, \{\langle !c, !d, ?c, ?d \rangle, \langle !c, !d, ?c, ?d \rangle \}
\]
In any case, since the parallel composition operator preserves the \( Nf \)-property when the set \( L \) of low level events and the set \( H \) of high level events are assumed to be all events that can occur on two distinct lifelines, we know that additional potential nondeterministic alternatives that arise during composition will not violated the \( Nf \)-property.

6 Related Work

This paper builds on [27] and in particular [28], where the authors showed how to preserve security properties under the notion of refinement (by underspecification) given in Sect. 2. A notion of security preserving transformations (similar to translation) was also defined. However, the security framework in [28] was based on [15], and composition was not considered. The main contributions of this paper are the security framework (Sect. 3), and conditions under which security properties are preserved under translation (Sect. 4) and composition (Sect. 5).

The notion of secure information flow was first introduced by Sutherland [29] as a generalization of the notion of non-interference [4]. Several such generalizations have later been proposed (see e.g. [18, 21, 31]) that again have led to the development of frameworks in which secure information flow properties can be represented in a uniform way and compared [1, 2, 15, 20, 22, 31]. Of these, the framework proposed by Mantel [15], is most similar to ours. Indeed, the modular way of constructing security properties from basic security predicates was introduced by Mantel [15], and the schema we propose for specifying such predicates is similar to Mantel’s schema [15]. The main difference between our schema and the one proposed by Mantel is that ours is based on relations, whereas the one proposed by Mantel is not. The use of relations makes it easier to propose conditions w.r.t. translation and composition.

Our framework is, at first glance, similar to the so-called generalized unwinding condition [2]. The difference is that the unwinding condition is specified in terms of a relation on processes in the setting of process algebras. Our framework is defined in terms of relations on traces in a trace based model. In general, unwinding conditions demand properties on events (actions) rather than traces. This results in proof obligations that are easier to handle, but yields a less abstract definition of security than what we may define in our framework. Our framework exploits the distinction of underspecification and unpredictability, but we are not aware of any unwinding conditions in which this is done.
The work that is most similar to ours is the work of Santen et al. [8, 26, 24, 25]. This is because the notions of information flow security, underspecification, data refinement (a notion similar to translation), and composition are all considered. Their notion of refinement is based on CSP refinement notions together with a relation from concrete events to abstract events. This notion of refinement is similar to our notion of refinement by underspecification modulo translation. However, our notion of translation is more general than the data refinement considered by Santen et al. because we map (abstract) traces to (concrete) trace sets as opposed to (abstract) events to (concrete) event sets. Without this generality, we would not have been able to characterize the transformation of the example in Sect. 4.1 where abstract events are mapped to several concrete traces. Another difference is that the work of Santen et al. considers probabilistic security properties as opposed to possibilistic security properties as we do. Consequently the semantic model of specifications and the security framework considered by Santen et al. are substantially different from the semantics and security framework of this paper.

Secure information flow properties in relation to data refinement also previously considered in [10, 5]. However, Graham-Cumming and Sanders [5] consider a less general notion of security than we do, and they only consider refinements of internal states of a system but not of the input and output data. The notion of data refinement presented by Hutter [10] is quite general, but refinement of underspecification is not considered. Another difference between Hutter’s paper [10] and our work is that Hutter’s specification model is based on so-called configuration structures and his security framework is different from ours.

There are a number of papers addressing the compositionality of secure information flow properties [2, 9, 19, 17, 20, 30]. [19, 30] focus on particular properties without considering a general framework of security properties. Hinton [9] proposes a notion of so-called emergent behavior and emergent properties as a way of analyzing composability of security properties. However, the treatment is rather informal in that no formal definition of a notion of composition nor formal conditions under which security properties are composable is given. [2, 17, 20] address compositionality in the light of a general framework for security properties. However, Mantel [17] and McLean [20] restricts attention to particular notions of composition. This is not the case for Bossi et al. [2], which handles compositionality of security properties in a manner that is similar to our approach. The main difference is that (1) the security framework of Bossi et al. [2] is based on the generalized unwinding condition in a process algebra, which as noted above, differs from the framework we propose in this paper, and (2) the distinction of unpredictability and underspecification is not made by Bossi et al.

Jacob is the first person that we are aware of to show that secure information flow properties are not preserved by the standard notion of refinement [11]. Since then, a large number of papers have investigated the relationship of information flow security and refinement of underspecification. These can be classified into two categories: those that strengthen the notion of refinement, and those that strengthen the notion of security, e.g. by closing the security properties under refinement. Our work is in line with the latter approach. Other works that follows this approach are [13, 14]. In both papers, the security properties are explicitly closed under refinement. The authors of the papers are able to do this without making the security properties unreasonably strong be-
cause the notion of refinement considered prohibits the refinement of *external nondeterminism* (which is determined by the environment); only refinement of *internal nondeterminism* (which is determined by the system) is allowed. The notion of refinement considered in this paper is more general because it both allows external nondeterminism to be refined and it allows refinement of internal nondeterministic alternatives to constrained by specifying these as explicit alternatives. This generality is useful, particularly when input totality is not assumed. For instance, the example of Sect. 5.1 in which we refine the communication medium would not have been possible without being able to refine external nondeterminism.

Related work that ensures security preservation under refinement by strengthening the notion of refinement (instead of the security properties) are [1, 3, 16]. Bossi et. al. [3] present sufficient conditions with which to check that a given refinement preserves information flow properties. Apart from the fact that they do not strengthen their notion of security, their work differs from ours in that they consider specifications expressed in a process calculus, and that they focus attention to bisimulation-based properties. Mantel [16] presents refinement operators that can be used to check or modify refinements such that security is preserved. However, according to Heisel et. al. [8], the refinement operators may lead to concrete specifications that are practically hard to implement, because the changes in the refinement they induce are hard to predict and may not be easy to realize in an implementation. Alur et. al. [1] presents a notion of secrecy preserving refinement. The underlying idea is that an implementation leaks a secret only when the specification also leaks it. Apart from the fact that they do not strengthen their notion of security, their approach differs from ours in that they consider a different security framework and their specification notion is based on labeled transition systems.

### 7 Conclusions and Future Work

We have presented a framework that supports an incremental and modular development process for secure software systems. The framework supports the preservation of information flow properties under a series of successive refinement, transformation, and composition steps.

In [28], the authors showed how to preserve security properties under the notion of refinement (by underspecification) given in Sect. 2. A notion of security preserving transformations (similar to translation) was also defined. However, the security framework in [28] was based on [15], and composition was not considered. The main contributions of this paper are the security framework (Sect. 3), and conditions under which security properties are preserved under translation (Sect. 4) and composition (Sect. 5).

The emphasis of this paper has been on simplicity and generality in a semantic setting. In future work, we will address syntactic notions of transformation. Eventually, we would like to develop a computerized tool that checks whether transformations violate security.
References


REFERENCES


A Proofs

A.1 System specifications

Theorem 1  If specification $\Omega'$ is a refinement of specification $\Omega$ and system $\Phi$ is in compliance with $\Omega'$, then $\Phi$ is in compliance with $\Omega$. Formally,

$$(\Omega \Rightarrow \Omega' \land \Omega' \Rightarrow \Phi) \implies \Omega \Rightarrow \Phi$$

Proof of Theorem 1

Proof: $$(\Omega \Rightarrow \Omega' \land \Omega' \Rightarrow \Phi) \implies \Omega \Rightarrow \Phi$$

(1) 1. Assume: $1.1. \Omega \Rightarrow \Omega'$
2. $\Omega' \Rightarrow \Phi$

Prove: $\Omega \Rightarrow \Phi$

(2) 1. ($t \in \Phi \implies \exists \phi \in \Omega : t \in \phi \land (\phi \in \Omega \implies \exists t \in \Phi : t \in \phi$)

(3) 1. Assume: $3.1. t \in \Phi$

Prove: $\exists \phi \in \Omega : t \in \phi$

(4) 1. Choose $\phi' \in \Omega'$ such that $t \in \phi'$

Proof: By assumption 1.2, assumption 3.1, and definition of compliance (Def. 3).

(4) 2. Choose $\phi \in \Omega$ such that $\phi' \subseteq \phi$

Proof: By (4) 1, assumption 1.1, and definition of refinement (Def. 4).

(4) 3. Q.E.D.

Proof: By (4) 1 and (4) 2.

(3) 2. Assume: $3.1. \phi \in \Omega$

Prove: $\exists t \in \Phi : t \in \phi$

(4) 1. Choose $\phi' \in \Omega'$ such that $\phi' \subseteq \phi$

Proof: By assumption 3.1, assumption 1.1, and definition of refinement (Def. 4).
A PROOFS

⟨4⟩2. Choose \( t \in \Phi \) such that \( t \in \phi' \)

**Proof:** By (4)1, assumption 1.2, and definition of compliance (Def. 3).

⟨4⟩3. Q.E.D.

**Proof:** By (4)1 and (4)2.

⟨3⟩3. Q.E.D.

**Proof:** By ⟨3⟩1 and ⟨3⟩2.

⟨2⟩2. Q.E.D.

**Proof:** By ⟨2⟩1 and definition of compliance (Def. 3).

⟨1⟩2. Q.E.D.

**Proof:** By ⟨1⟩1.

### A.2 Security framework for systems

**Theorem 3** The following equivalences hold:

\[
\begin{align*}
\text{Re}(\Phi) & \iff \text{Re}'(\Phi) \\
\text{Ri}(\Phi) & \iff \text{Ri}'(\Phi) \\
\text{Ihai}(\Phi) & \iff \text{Ihai}'(\Phi) \\
\text{Iae}(\Phi) & \iff \text{Iae}'(\Phi)
\end{align*}
\]

**Proof of Theorem 3** By Lemma 3.1, Lemma 3.2, Lemma 3.3, and Lemma 3.4.

**Lemma 3.1** The equivalence \( \text{Re}(\Phi) \iff \text{Re}'(\Phi) \) holds.

**Proof of Lemma 3.1**

**Assume:** 1.1 \( \text{Re}'(\Phi) \)

**Prove:** \( \text{Re}(\Phi) \)

⟨1⟩1. **Assume:** 1.1 \( \text{Re}'(\Phi) \)

**Prove:** \( \text{Re}(\Phi) \)

⟨1⟩2. **Assume:** 1.1 \( \text{Re}'(\Phi) \)

**Prove:** \( \text{Re}(\Phi) \)

⟨2⟩1. **Assume:** 1.1 \( \text{Re}'(\Phi) \)

**Prove:** \( \text{Re}(\Phi) \)

⟨2⟩2. **Assume:** 1.1 \( \text{Re}'(\Phi) \)

**Prove:** \( \text{Re}(\Phi) \)

⟨2⟩3. **Assume:** 1.1 \( \text{Re}'(\Phi) \)

**Prove:** \( \text{Re}(\Phi) \)

⟨2⟩4. **Assume:** 1.1 \( \text{Re}'(\Phi) \)

**Prove:** \( \text{Re}(\Phi) \)

⟨4⟩1. **Assume:** 1.1 \( \text{Re}'(\Phi) \)

**Prove:** \( \text{Re}(\Phi) \)

⟨4⟩2. **Assume:** 1.1 \( \text{Re}'(\Phi) \)

**Prove:** \( \text{Re}(\Phi) \)

⟨4⟩3. **Assume:** 1.1 \( \text{Re}'(\Phi) \)

**Prove:** \( \text{Re}(\Phi) \)

⟨4⟩4. **Assume:** 1.1 \( \text{Re}'(\Phi) \)

**Prove:** \( \text{Re}(\Phi) \)

⟨4⟩5. **Assume:** 1.1 \( \text{Re}'(\Phi) \)

**Prove:** \( \text{Re}(\Phi) \)

⟨4⟩6. **Assume:** 1.1 \( \text{Re}'(\Phi) \)

**Prove:** \( \text{Re}(\Phi) \)
Proof: By (3)1 and (3)2.
(2)2. Q.E.D.

Proof: By (2)1 and definition of Re' (Sect. 3.2).
(1)3. Q.E.D.

Proof: By (1)1 and (1)2.

Lemma 3.2 The equivalence $R_i(\Phi) \iff R_i'(\Phi)$ holds.

Proof of Lemma 3.2 The proof is the same as for Lemma 3.1, except that all occurrences of $H$ in the proof must be replaced by $HI$.

Lemma 3.3 The equivalence $I_{hai}(\Phi) \iff I_{hai}'(\Phi)$ holds.

Proof of Lemma 3.3

(1)1. Assume: 1.1 $I_{hai}'(\Phi)$

Prove: $I_{hai}(\Phi)$

(2)1. Assume: 2.1 $s, t \in \Phi$

$2.2 e \in HI$

$2.3 \alpha|_{HI} = \emptyset$ for some $\alpha \in T$

$2.4 t = \beta \sim \alpha$ for some $\beta \in T$

$2.5 \gamma'|_{HI} = \beta|_{HI}$ for some $\gamma' \in T$

$2.6 s = \gamma' \sim (e)$

Prove: $\exists u \in \Phi : u \sim_l t \land s|_{HI} = u|_{HI}$

(3)1. $R_{I_{hai}}(\Phi, t, \alpha, \beta, e)$

Proof: By assumptions 2.1 - 2.6 and definition of $R_{I_{hai}}$ (Table 1 in Sect. 3.2).

(3)2. Choose $u \in \Phi$ and $\alpha', \beta' \in T$ such that $u \sim_l t$, $u = \beta' \sim (e) \sim \alpha'$, $\beta'|_{JI} = \beta|_{JI}$, and $\alpha'|_{JI} = \alpha|_{JI}$

Proof: By (3)1, assumptions 1.1, definition of $I_{hai}$ (Sect. 3.2).

(3)3. $s|_{HI} = u|_{HI}$

Prove:

$\gamma'|_{HI} = \beta|_{HI}$ By assumption 2.5

$\gamma'|_{HI} = \beta'|_{HI}$ By (3)2

$(\gamma' \sim (e))|_{HI} = (\beta' \sim (e))|_{HI}$ By definition of $|$ $\gamma' \sim (e))|_{HI} = (\beta' \sim (e) \sim \alpha')|_{HI}$ By assumption 2.3 and (3)2

$s|_{HI} = u|_{HI}$ By assumption 2.6 and (3)2

(3)4. Q.E.D.

Proof: By (3)2, (3)3.

(2)2. Q.E.D.

Proof: By definition of $I_{hai}$ (Sect. 3.2).

(1)2. Assume: 1.1 $I_{hai}(\Phi)$

Prove: $I_{hai}'(\Phi)$

(2)1. Assume: 2.1 $t \in \Phi$, $\alpha, \beta \in T$, and $e \in E$

$2.2 e \in HI$

$2.3 \alpha|_{HI} = \emptyset$

$2.4 t = \beta \sim \alpha$

$2.5 \gamma'|_{HI} = \beta|_{HI}$ for some $\gamma' \in T$

$2.6 \gamma' \sim (e) \in \Phi$
Prove: \( \exists u \in \Phi, \alpha', \beta' \in \mathcal{T}: u \sim t \land u = \beta' \sim \langle e \rangle \sim \alpha' \land \beta'|_{\mathcal{L} \cup \mathcal{H}_T} = \beta|_{\mathcal{L} \cup \mathcal{H}_T} \land \alpha'|_{\mathcal{L} \cup \mathcal{H}_T} = \alpha|_{\mathcal{L} \cup \mathcal{H}_T} \)

\( \langle 3 \rangle \). 1. Choose \( s \in \Phi \) such that \( s = \gamma' \sim \langle e \rangle \)

Proof: By assumption 2.6.

\( \langle 3 \rangle \). 2. \( s \xrightarrow{\text{Ihai}} t \)

Proof: By assumptions 2.1 - 2.6, (3)1 and definition of \( \text{Ihai} \) (Sect. 3.2).

\( \langle 3 \rangle \). 3. Choose \( u \in \Phi \) such that \( u \sim t \) and \( u|_{\mathcal{H}_T} = u|_{\mathcal{H}_T} \)

Proof: By assumption 1.2, (3)2 and definition of \( \text{Ihai} \) (Sect. 3.2).

\( \langle 3 \rangle \). 4. \( u|_{\mathcal{H}_T \cup \mathcal{L}} = (\beta \sim \langle e \rangle \sim \alpha)|_{\mathcal{H}_T \cup \mathcal{L}} \)

Proof: By (3)1, assumption 1.1, and definition of \( \text{Ihai} \) (Sect. 3.2).

\( \langle 3 \rangle \). 5. Q.E.D.

Proof: By (3)4.

(2). Q.E.D.

Proof: By definition of \( \text{Ihai}' \) (Sect. 3.2).

(1). Q.E.D.

Proof: By (1)1 and (1)2.

**Lemma 3.4** The equivalence \( \text{IAE}(\Phi) \Leftrightarrow \text{IAE}'(\Phi) \) holds.

**Proof of Lemma 3.4**

Prove: \( \text{IAE}(\Phi) \Leftrightarrow \text{IAE}'(\Phi) \)

\( \langle 1 \rangle \). Assume: 1.1 \( \text{IAE}'(\Phi) \)

Prove: \( \text{IAE}(\Phi) \)

\( \langle 2 \rangle \). Assume: 2.1 \( s, t \in \Phi \)

\( 2.2 \ e \in \mathcal{H} \)

\( 2.3 \ \alpha|_{\mathcal{H}} = \langle \rangle \) for some \( \alpha \in \mathcal{T} \)

\( 2.4 \ t = \beta \sim \alpha \) for some \( \beta \in \mathcal{T} \)

\( 2.5 \ s = \beta \sim \langle e \rangle \)

Proof: \( \exists u \in \Phi : u \sim t \land h(s) = h(u) \)

\( \langle 3 \rangle \). \( R_{\text{IAE}}'(\Phi, t, \alpha, \beta, e) \)

Proof: By assumptions 2.1 - 2.5 and definition of \( R_{\text{IAE}}' \) (Sect. 3.2).

\( \langle 3 \rangle \). 2. Choose \( u \in \Phi \) such that \( t \sim u \) and \( u = \beta \sim \langle e \rangle \sim \alpha \)

Proof: By (3)1, assumption 1.1, and definition of \( \text{IAE}' \) (Sect. 3.2).

\( \langle 3 \rangle \). 3. \( h(s) = h(u) \)

Proof: By (3)2, assumption 2.5, assumption 2.3, and definition of \( h \) (Sect. 3.2).

\( \langle 3 \rangle \). 4. Q.E.D.

Proof: By (3)2 and (3)3.

(2). Q.E.D.

Proof: By definition of \( \text{IAE} \) (Sect. 3.2).

\( \langle 1 \rangle \). Assume: 1.1 \( \text{IAE}(\Phi) \)

Prove: \( \text{IAE}'(\Phi) \)

\( \langle 2 \rangle \). Assume: 2.1 \( t \in \Phi, \alpha, \beta \in \mathcal{T}, \) and \( e \in \mathcal{E} \)

\( 2.2 \ e \in \mathcal{H} \)

\( 2.4 \ \alpha|_{\mathcal{H}} = \langle \rangle \)
2.5 \( t = \beta \sim \alpha \)
2.6 \( \beta \sim \langle e \rangle \in \Phi \)

**Prove:** \( \exists u \in \Phi : u \sim t \land u = \beta \sim \langle e \rangle \sim \alpha \)

**Proof:** By assumption 2.6.

\(<3>2\). Choose \( s \in \Phi \) such that \( s = \beta \sim \langle e \rangle \)

**Proof:** By assumption 2.1 - 2.6, \(<3>1\) and definition of \( \tau_{\text{IAE}} \) (Sect. 3.2).

\(<3>3\). Choose \( u \in \Phi \) such that \( u \sim t \) and \( h(s) = h(u) \)

**Proof:** By \(<3>2\), assumption 1.1, and definition of IAE (Sect. 3.2).

\(<3>4\). \( u = \beta \sim \langle e \rangle \sim \alpha \)

1. \( h(\beta \sim \langle e \rangle) = h(u) \)

**Proof:** By \(<3>1\) and \(<3>3\).

2. \( (\beta \sim \alpha) \mid L = u \mid L \)

**Proof:** By assumption 2.5, \(<3>3\), and definition of \( | \).

3. \( \beta \subseteq u \)

**Proof:** By \(<4>1\), assumption 2.2, and definition of \( h \), there must be a prefix of \( u \) that is identical to \( \beta \) when we replace all low-level events in the two traces by the dummy event \( \sqrt{.} \). By \(<4>2\), that prefix must also contain exactly the same low-level events that are in \( \beta \). Hence, the prefix is equal to \( \beta \).

4. \( \beta \sim \langle e \rangle \subseteq u \)

**Proof:** By assumption 2.2, \(<4>1\), \(<4>3\), and definition of \( h \).

5. Choose \( \alpha' \in \mathcal{T} \) such that \( u = \beta \sim \langle e \rangle \sim \alpha' \)

**Proof:** By \(<4>4\) and definition of \( \subseteq \).

6. \( \alpha' = \alpha \)

**Proof:** If \( \alpha' \neq \alpha \), then (a) \( \alpha' \) must contain a high-level event (by assumption 2.4) or (b) \( \alpha' \) must differ from \( \alpha \) on the low-level events. (a) cannot hold due to \(<4>1\) and definition of \( h \), and (b) cannot hold due to \(<4>2\) and definition of \( | \). Therefore \( \alpha' = \alpha \).

7. **Q.E.D.**

**Proof:** By \(<4>5\) and \(<4>6\).

\(<3>5\). **Q.E.D.**

**Proof:** By \(<3>1\), \(<3>3\), and \(<3>4\).

\(<2>2\). **Q.E.D.**

**Proof:** By definition IAE' (Sect. 3.2).

\(<1>3\). **Q.E.D.**

**Proof:** By \(<1>1\) and \(<1>2\).

### A.3 Security framework for specifications

**Theorem 4** \( \text{Bsp}_{\tau_{\text{thl}}} \) is preserved by refinement for arbitrary restriction relation \( \tau \), high-level relation \( h \), and low-level equivalence relation \( t \):

\[ \Omega \rightarrow \Omega' \land \text{Bsp}_{\tau_{\text{thl}}} (\Omega) \implies \text{Bsp}_{\tau_{\text{thl}}} (\Omega') \]

**Proof of Theorem 4**

**Assume:**
1. \( \Omega \rightarrow \Omega' \)
2. \( \text{Bsp}_{\tau_{\text{thl}}} (\Omega) \)

**Prove:** \( \text{Bsp}_{\tau_{\text{thl}}} (\Omega') \)
A PROOFS

(1). Assume: 1.1 \(s \xrightarrow{r} t\) for some \(s, t \in \overline{\Omega}\)

Prove: \(\exists \phi' \in \Omega': \{s\} \xrightarrow{h} \phi' \sim_l \{t\}\)

(2). \(s, t \in \overline{\Omega}\)

Proof: By assumption 1.1, assumption 1, and definition of \(\xrightarrow{r}\) (Def. 4).

(2). Choose \(\phi \in \Omega\) such that \(\{s\} \xrightarrow{h} \phi \sim_l \{t\}\)

Proof: By assumption 1.1, (2).1, assumption 2, and definition of Bsp’s (Def. 8).

(3). Choose \(\phi' \in \Omega'\) such that \(\phi' \subseteq \phi\)

Proof: By (2).2, assumption 1, and definition of \(\xrightarrow{r}\) (Def. 4).

(4). \(\{s\} \xrightarrow{h} \phi' \sim_l \{t\}\)

Proof: By (2).2, (2).3 and definition of \(\xrightarrow{r}\) on trace sets (see Sect. 3.3).

(5). Q.E.D.

Proof: By (2).4.

(1). Q.E.D.

Proof: By (1).1 and definition of Bsp’s (Def. 8).

Theorem 5 Let \(\Phi\) be a system, and let \(\Omega\) be the specification obtained from \(\Phi\) by creating a new obligation for each trace in \(\Phi\), i.e. \(\Omega = \{\{t\} | t \in \Phi\}\). Then \(\Omega\) satisfies the basic security predicate \(\text{Bsp}_{\text{thl}}\) (of Def. 8) if and only if \(\Phi\) satisfies \(\text{Bsp}_{\text{thl}}\) (of Def. 5), i.e.,

\[\text{Bsp}_{\text{thl}}(\Phi) \iff \text{Bsp}_{\text{thl}}(\{\{t\} | t \in \Phi\})\]

Proof of Theorem 5

Assume: 1. \(\Omega = \{\{t\} | t \in \Phi\}\)

Prove: \(\text{Bsp}_{\text{thl}}(\Phi) \iff \text{Bsp}_{\text{thl}}(\Omega)\)

(1). Assume: 1. \(\text{Bsp}_{\text{thl}}(\Phi)\)

Prove: \(\text{Bsp}_{\text{thl}}(\Omega)\)

(2). Assume: 2.1 \(s \xrightarrow{r} t\) for some \(s, t \in \overline{\Omega}\)

Prove: \(\exists \phi \in \Omega: \{s\} \xrightarrow{h} \phi \sim_l \{t\}\)

(3).1. Choose \(u \in \Phi\) such that \(s \xrightarrow{h} u \sim_l t\)

Proof: By assumption 1, assumption 1.1, assumption 2.1 and definition of Bsp (Def. 5).

(3).2. Choose \(\phi \in \Omega\) such that \(\phi = \{u\}\)

Proof: By assumption 1.

(3).3. Q.E.D.

Proof: By (3).1 and (3).2 and definition of \(\xrightarrow{r}\) on trace sets (Sect.3.3).

(2). Q.E.D.

Proof: By definition of Bsp (Def. 8).

(1).2. Assume: 1. \(\text{Bsp}_{\text{thl}}(\Omega)\)

Prove: \(\text{Bsp}_{\text{thl}}(\Phi)\)

(2).1. Assume: 2.1 \(s \xrightarrow{r} t\) for some \(s, t \in \Phi\)

Prove: \(u \in \Phi: s \xrightarrow{h} u \sim_l t\)

(3).1. Choose \(\{u\} \in \Omega\) such that \(\{s\} \xrightarrow{h} \{u\} \sim_l \{t\}\)

Proof: By assumption 1, assumption 1.1, assumption 2.1 and definition of Bsp (Def. 8).

(3).2. Q.E.D.
Proof: By (3)1 and definition of \( \rightarrow \) on trace sets (Sect.3.3).

(2)2. Q.E.D.

Proof: By definition of BSP (Def. 8).

(1)3. Q.E.D.

Proof: By (1)1 and (1)2.

A.4 Translation

Theorem 6 Specification \( F(\Omega) \) is secure w.r.t. \( \text{BSP}_{r,t_{r,t_1}} \) if \( \Omega \) is secure w.r.t. \( \text{BSP}_{t_{r,t_1}} \) and \( F \) is a translation that satisfies the following conditions for all \( f_1, f_2 \in F, s, t \in \Omega, \phi \in \Omega, \text{ and } s' \in f_1(s), \text{ and } t' \in f_2(t) \)

\[
\wedge_{\phi} s' \xrightarrow{\cdot_{r}} t' \implies s \xrightarrow{\cdot_{t}} t
\]

\[
\{s\} \xrightarrow{\cdot_{r}} \phi \sim t \implies \exists f \in F: \{s'\} \xrightarrow{\cdot_{t}} f(\phi) \sim \{t'\}
\]

Proof of Theorem 6

Assume:

1. \( F \subseteq T \rightarrow (P(T') \setminus \emptyset) \)
2. \( \text{BSP}_{t_{r,t}}(\Omega) \) for arbitrary \( \Omega \in P(P(T)) \)
3. \( \forall f_1, f_2 \in F, s, t, s' \in f_1(s), t' \in f_2(t) : s' \xrightarrow{r_{\cdot}} l_{\cdot} t' \implies s \xrightarrow{t_{\cdot}} t \)
4. \( \forall f_1, f_2 \in F, s, t, s' \in f_1(s), t' \in f_2(t) : \{s\} \xrightarrow{r_{\cdot}} \phi \sim \{t\} \)

Prove: \( \text{BSP}_{r,t_{r,t}}(F(\Omega)) \)

(1)1. Assume: 1.1 \( s' \xrightarrow{r_{\cdot}} t' \) for some \( s', t' \in F(\Omega) \)

Prove: \( \exists \phi' \in F(\Omega) : \{s'\} \xrightarrow{t_{\cdot}} \phi' \sim \{t'\} \)

(2)1. Choose \( f_1, f_2 \in F \) and \( s, t \in \Omega \) such that \( s' \in f_1(s) \) and \( t' \in f_2(t) \)

Proof: By assumption 1, 1.1, and definition of translation (Def. 9).

(2)2. \( s \xrightarrow{t_{\cdot}} t \)

Proof: By (2)1, assumption 1.1, and assumption 3.

(2)3. Choose \( \phi \in \Omega \) such that \( \{s\} \xrightarrow{r_{\cdot}} \phi \sim \{t\} \)

Proof: By assumption 2, (2)2, and definition of BSP’s (Def. 8).

(2)4. Choose \( f \in F \) and \( \phi' \in F(\Omega) \) such that \( \phi' = f(\phi) \) and \( \{s'\} \xrightarrow{t_{\cdot}} \phi' \sim \{t'\} \)

Proof: By (2)3, assumption 1, definition of translation (Def. 9), and assumption 4.

(2)5. Q.E.D.

Proof: By (2)4.

(1)2. Q.E.D.

Proof: By definition of BSP’s (Def. 8).

A.5 Composition

Theorem 7 \( \Omega_1 \circ \Omega_2 \) is secure w.r.t. \( \text{BSP}_{t_{r,t_1}} \) if \( \Omega_1 \) is secure w.r.t. \( \text{BSP}_{t_{r,t_1}} \), \( \Omega_2 \) is secure w.r.t. \( \text{BSP}_{r_{t_{r,t_1}}}, \) and \( \circ \) is a composition operator that satisfies
the following conditions for all $\phi_1, \phi_2 \in \Theta$, $s_1, t_1 \in \widehat{\Omega}_1$, $\phi_1 \in \Omega_1$, $s_2, t_2 \in \widehat{\Omega}_2$, $\phi_2 \in \Omega_2$, $s \in (s_1 \circ s_2)$, $t \in (t_1 \circ t_2)$

$$s \overset{\tau}{\rightarrow} t \implies (s_1 \overset{\tau_1}{\rightarrow} t_1 \land s_2 \overset{\tau_2}{\rightarrow} t_2)$$

$$\{s_1\} \overset{h_1}{\rightarrow} \phi_1 \sim_{\tau_1} \{t_1\} \land \{s_2\} \overset{h_2}{\rightarrow} \phi_2 \sim_{\tau_2} \{t_2\} \implies \exists \phi \in \Theta : \{s\} \overset{h}{\rightarrow} (\phi \circ \phi_2) \sim_{\tau_1} \{t\}$$

Proof of Theorem 7

Assume: 1. $\Theta \subseteq T \times T \rightarrow (\mathcal{P}(T) \setminus \emptyset)$

2. Bsp$_{h,1,1}(\Omega_1)$ for some $\Omega_1 \in \mathcal{P}(\mathcal{P}(T))$

3. Bsp$_{h,1,1}(\Omega_2)$ for some $\Omega_2 \in \mathcal{P}(\mathcal{P}(T))$

4. $\forall \phi_1, \phi_2 \in \Theta$, $s_1, t_1 \in \widehat{\Omega}_1$, $s_2, t_2 \in \widehat{\Omega}_2$, $s \in (s_1 \circ s_2)$, $t \in (t_1 \circ t_2)$:

$$s \overset{\tau}{\rightarrow} t \implies (s_1 \overset{\tau_1}{\rightarrow} t_1 \land s_2 \overset{\tau_2}{\rightarrow} t_2)$$

5. $\forall \phi_1, \phi_2 \in \Theta$, $s_1, t_1 \in \widehat{\Omega}_1$, $\phi_1 \in \Omega_1$, $s_2, t_2 \in \widehat{\Omega}_2$, $\phi_2 \in \Omega_2$, $s \in (s_1 \circ s_2)$, $t \in (t_1 \circ t_2)$:

$$\{s_1\} \overset{h_1}{\rightarrow} \phi_1 \sim_{\tau_1} \{t_1\} \land \{s_2\} \overset{h_2}{\rightarrow} \phi_2 \sim_{\tau_2} \{t_2\} \implies \exists \phi \in \Theta : \{s\} \overset{h}{\rightarrow} (\phi \circ \phi_2) \sim_{\tau_1} \{t\}$$

Proof: Bsp$_{h \circ \tau_1}(\Omega_1 \circ \Omega_2)$

(1.1) Assume: 1.1 $s \overset{\tau}{\rightarrow} t$ for some $s, t \in \widehat{\Omega}_1 \circ \Omega_2$

Proof: By assumption 1.1, assumption 1, and definition of composition (Def. 10).

(2.1) Choose $\phi_1, \phi_2 \in \Theta$, $s_1, t_1 \in \widehat{\Omega}_1$, and $s_2, t_2 \in \widehat{\Omega}_2$ such that $s \in (s_1 \circ s_2)$ and $t \in (t_1 \circ t_2)$

Proof: By assumption 1.1, assumption 1, and assumption 4.

(2.2) Choose $\phi_1 \in \Omega_1$ such that $\{s_1\} \overset{h_1}{\rightarrow} \phi_1 \sim_{\tau_1} \{t_1\}$

Proof: By assumption 2, (2.2), and definition of Bsp’s (Def. 8).

(2.3) Choose $\phi_2 \in \Omega_2$ such that $\{s_2\} \overset{h_2}{\rightarrow} \phi_2 \sim_{\tau_2} \{t_2\}$

Proof: By assumption 3, (2.2), and definition of Bsp’s (Def. 8).

(2.4) Choose $\phi \in \Theta$ such that $\{s\} \overset{h}{\rightarrow} \phi \sim_{\tau_1} \{t\}$

Proof: By (2.3), (2.4), definition of composition (Def. 10), and assumption 5.

(2.6) Q.E.D.

Proof: By (2.5).

(1.2) Q.E.D.

Proof: By definition of Bsp’s (Def. 8).
Chapter 12

Adherence Preserving Refinement of Trace-set Properties in STAIRS
Exemplified for Information Flow Properties and Policies

Fredrik Seehusen\textsuperscript{1,2}, Bjørnar Solhaug\textsuperscript{1,3}, Ketil Stølen\textsuperscript{1,2}\textsuperscript{*}

\textsuperscript{1} SINTEF ICT
\textsuperscript{2} Dep. of Informatics, University of Oslo
\textsuperscript{3} Dep. of Information Science and Media Studies, University of Bergen

Received: date / Revised version: date

Abstract

STAIRS is a formal approach to system development with UML 2.1 sequence diagrams that supports an incremental and modular development process. STAIRS is underpinned by denotational and operational semantics that have been proved to be equivalent. STAIRS is more expressive than most approaches with a formal notion of refinement. STAIRS supports a stepwise refinement process under which trace properties as well as trace-set properties are preserved. This paper demonstrates the potential of STAIRS in this respect, in particular that refinement in STAIRS preserves adherence to information flow properties as well as policies.

1 Introduction

The denotational semantics of STAIRS [9,33] is a formalization of the trace semantics for sequence diagrams that is informally described in the UML 2.1 standard [27]. A trace is a sequence of event occurrences (referred to as “events” in the following) ordered by time that describes an execution history, and the semantics of a sequence diagram is defined in the standard by a set of valid traces and a set of invalid traces.

A system implementation, i.e., an existing system, can be characterized by the set of traces representing all the possible runs of the system. Such a representation can serve as a basis for analysis as well as verification or falsification of different properties, e.g., safety and security, that the system is supposed to possess or not to possess. Property falsification can be classified into falsification on the basis of a single trace and falsification on the basis of a set of traces [25].

Properties that can be falsified on single traces are the kind of properties that were originally investigated by Alpern and Schneider [2,35] and include safety and liveness properties. Properties that can only be falsified on sets of traces are of the type that McLean [25] referred to as possibilistic properties and include information flow properties and permission rules of policies.

In system development with refinement, system documentation is organized in a hierarchical manner. The various specifications within the hierarchy are related by notions of refinement. Each refinement step adds details to the specification, making it more concrete and closer to an implementation. This facilitates efficiency of analysis and verification through abstraction. Moreover, it supports early discovery of design flaws.

STAIRS distinguishes between two kinds of non-determinism; namely, underspecification and inherent non-determinism. Underspecification is a feature of abstraction and means specifying several behaviors, each representing a potential alternative serving the same purpose. The fulfillment of only some of them (at least one) is acceptable for an implementation to be correct. The other kind of non-determinism is an explicit, inherent non-determinism that a correct implementation is required to possess. A simple example is the non-determinism between heads and tails in the specification of the game of tossing a coin. An implementation that offers only one of the alternatives is obviously not correct.

If an abstract system specification does not have the expressiveness to distinguish between these two kinds of non-determinism, refinement of trace-set properties is problematic [15,31]. Refinement will typically reduce underspecification and thereby reduce non-determinism. If there is no way to distinguish the non-determinism that can be reduced from the non-determinism that should be preserved, we have no guarantee that trace-set properties are preserved. In that case, although a trace-set property is verified to hold for the abstract specification, it may not hold for a system obtained by refinement from the abstract specification. The potential of trace-set properties not to be preserved under refinement is in [16] referred to as the refinement paradox.

\textsuperscript{*} Present address: Ketil.Stolen@sintef.no
The objective of this paper is twofold. Firstly, the paper presents the semantic foundation of STAIRS. The denotational semantics that captures both underspecification and inherent non-determinism is presented, along with a formal notion of refinement that preserves inherent non-determinism. Secondly and more importantly, the potential of the expressiveness of STAIRS to capture trace-set properties and preserve these under refinement is demonstrated within the domains of information flow security and policy based management, both of which require this expressiveness.

Generally, secure information flow properties provide a way of specifying security requirements, e.g., confidentiality, by selecting a set of domains, i.e., abstractions of real system entities such as users or files, and then restricting allowed flow of information between these domains. Many information flow properties are trace-set properties, and we show that such properties can be expressed within the STAIRS approach.

Policies define requirements to systems and are typically used for the management of security, networks, and services. A policy is defined as a set of policy rules. We show that some policies are trace-set properties and that they can be expressed in the STAIRS approach.

As illustrated by Fig. 1, we distinguish between system specifications (of which a system is a special case) on the one hand and properties/policies on the other hand. The notion of refinement relates different levels of abstractions on the system side and is therefore represented by the vertical arrows in Fig. 1. Adherence, on the other hand, is depicted by an arrow relating the left and right sides, and characterizes what it means for a system specification to fulfill a property/policy.

![Fig. 1 Preservation of adherence under refinement](image)

To the right in Fig. 1, \( S_1 \) denotes an initial, abstract system specification, whereas \( S_2 \) and \( S_3 \) denote refined and more concrete system specifications. We show that refinement as defined in STAIRS preserves adherence. Given that the specification \( S_1 \) adheres to \( P \), and \( S_2 \) is a refinement of \( S_1 \), it follows by adherence preservation that also \( S_2 \) adheres to \( P \), as illustrated by the upper left dashed arrow. The same is the case for refinement of \( S_3 \), resulting in the specification \( S_3 \). Moreover, the refinement relation is transitive, so if the development process begins with the specification \( S_1 \), and \( S_1 \) adheres to \( P \), then adherence is preserved under any number of refinement steps. Hence, also the specification \( S_3 \) adheres to \( P \), which motivates the dashed arrow from the uppermost adherence relation to the lowermost.

The structure of the paper is as follows. In the next section, we give an overview of the central concepts of the STAIRS approach and show how these are related to concepts of information flow security on the one hand and to concepts of policies on the other hand. In Section 3, we give an introduction to UML 2.1 sequence diagrams and the STAIRS denotational trace semantics. In Section 4, information flow properties are defined and formalized, information flow property adherence is defined, and it is shown that information flow property adherence is preserved under refinement. In Section 5, we introduce a syntax and a semantics for specifying policies in the setting of UML 2.1 sequence diagrams. We formally define the notion of policy adherence and show preservation of policy adherence under refinement. The results of the paper are related to other work in Section 6. Finally, in Section 7, we conclude. There is also an appendix with formal definitions of some notions and concepts that for the sake of readability are only defined informally in the main part of the paper.

2 Conceptual Overview

The UML class diagram in Fig. 2 depicts the most important concepts and conceptual relations of relevance to this paper. In STAIRS, a system specification is given by a UML 2.1 sequence diagram. Semantically, the meaning of a sequence diagram is captured by a non-empty set of interaction obligations. Each interaction obligation represents one inherent choice of system behavior. The variation over interaction obligations captures the inherent non-determinism that is required for trace-set properties to be expressed.

Each interaction obligation is defined by a pair of trace sets, one positive and one negative. The set of positive traces defines valid or legal alternatives, and may capture underspecification since a correct implementation is only required to offer at least one of these traces. The set of negative traces defines invalid or illegal alternatives, and a correct implementation of an interaction obligation will never produce any of the negative traces. Notice that the positive traces in one interaction obligation may be negative in other interaction obligations. Hence, the scope of a negative trace is the interaction obligation in question.

In the UML 2.1 standard, a sequence diagram defines both valid and invalid traces that correspond to the pos-
Adherence Preserving Refinement of Trace-set Properties in STAIRS

3 STAIRS

As background to the full treatment of trace-set properties and adherence, in this section, we give a more mathematical introduction to STAIRS with focus on the denotational semantics of sequence diagrams and the notion of refinement. We refer to [9,33] for the full treatment. See [21] for an equivalent operational semantics.

We first present the semantics of sequence diagrams without the STAIRS specific \texttt{xalt} operator that is used to specify inherent non-determinism. In the cases without the use of \texttt{xalt}, a single interaction obligation is sufficient. We then address the general situation with \texttt{xalt}, in which case the semantics of a sequence diagram becomes a set of interaction obligations. Finally, we define refinement.

3.1 Sequence Diagrams without \texttt{xalt}

STAIRS formalizes UML sequence diagrams, and thus precisely defines the trace semantics that is only informally described in the UML 2.1 standard. Sequence diagrams specify system behavior by showing how entities interact by the exchange of messages, where the behavior is described by traces.

A trace is a sequence of events ordered by time representing a system run. An event is either the transmission or the reception of a message. In the STAIRS denotational semantics, a message is given by a triple \((co, tr, re)\) of a message content \(co\), a transmitter \(tr\), and a receiver \(re\). The transmitter and receiver are lifelines. \(L\) denotes the set of all lifelines and \(M\) denotes the set of all messages. An event is a pair of a kind and a message, \((k, m)\) \(\in \{!, ?\} \times M\), where ! denotes that it is a transmit event and ? denotes that it is a receive event. By \(E\), we denote the set of all events, and \(T\) denotes the set of all traces.

The diagram \texttt{login} to the left in Fig. 3 is very basic and has only two events; the transmission of the message \texttt{login(id)} on the user lifeline and the reception of the same message on the server lifeline. The transmit event must obviously occur before the reception event. The diagram \texttt{login} therefore describes the single trace \([l!, r?]\) of these two events, where \(l\) is a shorthand for the message. Throughout the paper we will represent a message by the first letter in the message content for simplicity’s sake.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{sequence_diagram.png}
\caption{Basic sequence diagrams}
\end{figure}

The diagram \texttt{request} to the right in Fig. 3 shows the sequential composition of the transmission and reception of the two messages \(l\) and \(r\). The order of the events on each lifeline is given by their vertical positions, but the two lifelines are independent. This corresponds to so-called weak sequencing. Weak sequencing defines a partial ordering of the events of the diagram, and requires that events on the same lifeline are ordered sequentially, and that the transmission of a message is ordered before its reception. For traces \(t_1\) and \(t_2\), \(t_1 \succeq t_2\) is the
set of traces obtained by their weak sequencing.\footnote{For a formal definition of the weak sequencing operator \( \succcurlyeq \), see Appendix A.2.} The weak sequencing in the diagram \textit{request}, for example, is captured by

\[
\langle \text{User}, \text{Server} \rangle, \langle \text{User}, \text{Server} \rangle, \langle \text{User}, \text{Server} \rangle, \langle \text{User}, \text{Server} \rangle \}
\]

The transmission of \textit{l} is the first event to occur, but after that, both the reception of \textit{l} and the transmission of \textit{r} may occur. The \( \succcurlyeq \) operator is lifted to sets of traces \( T_1 \) and \( T_2 \) in a pointwise manner as follows:

\[
T_1 \succcurlyeq T_2 \overset{\text{def}}{=} \bigcup_{(t_1, t_2) \in T_1 \times T_2} (t_1 \succcurlyeq t_2)
\]

Notice that if \( T_1 \) or \( T_2 \) is \( \emptyset \), the result is also \( \emptyset \).

In the diagram \textit{request’} in Fig. 4, we employ a construct referred to as an interaction use in the UML standard. The ref operator takes a sequence diagram as operand, which in this case is the diagram \textit{login} to the left in Fig. 3. An interaction use is merely shorthand for the contents of the referred diagram, but is useful as it allows the reuse of the same specification in several different contexts. As a result, the diagram \textit{request’} shows the weak sequencing of the transmission and reception of the two messages \textit{l} and \textit{r}. The diagram \textit{request’} is hence semantically equivalent to the diagram \textit{request} of Fig. 3.

Generally, a sequence diagram specifies both valid and invalid behavior that is semantically captured by a pair \( (p, n) \) of positive and negative traces, respectively. Such pairs are referred to in STAIRS as interaction obligations. Traces not specified as positive or negative, i.e., \( T \setminus (p \cup n) \), are referred to as inconclusive. For both diagrams in Fig. 3, the set of negative traces is the empty set \( \emptyset \) since no behavior is specified as negative.

The \texttt{neg} construct may be used to specify behavior as negative, as exemplified in Fig. 6. This diagram specifies that following a login failure (positive scenario), the user is not allowed to retrieve documents from the server (negative scenario). The diagram is thus the weak sequencing of a positive and a negative scenario. In order to capture this formally, we lift the weak sequencing operator from sets of traces to interaction obligations as follows.

\[
(p_1, n_1) \succcurlyeq (p_2, n_2) \overset{\text{def}}{=} (p_1 \succcurlyeq p_2, (n_1 \succcurlyeq n_2) \cup (n_1 \succcurlyeq n_2))
\]

Notice that any combination with a negative scenario is defined as negative.

To see what this definition means for the diagram in Fig. 6, consider first the negative scenario, i.e., the interaction within the scope of the \texttt{neg} operator. It defines exactly one negative trace; namely, \( \langle \text{i}, \text{r}, \text{f}, \text{t}, \text{d} \rangle \). It also defines one positive trace; namely, the empty trace \( \langle \rangle \), indicating that skipping the behavior specified by the negative scenario is valid. The interaction obligation corresponding to the negative scenario is therefore \( \langle \langle \rangle \rangle \), \( \langle \langle \text{l}, \text{r}, \text{f}, \text{t}, \text{d} \rangle \rangle \).
This yields an interaction obligation consisting of one positive trace and one negative trace:

\[
\]

The positive traces of the sequence diagram request in Fig. 3 are equivalent in the sense that they both represent a valid way of fulfilling the behavior as described by the diagram. In STAIRS, this kind of non-determinism, often referred to as underspecification, may also be expressed directly by the alt operator. This is exemplified in the diagram sendDoc in Fig. 7, where two alternative ways of sending data from the server to the user is described. Both operands of the alt operator represent adequate alternatives of fulfilling the intended behavior of sending two documents to the user. It is left to the implementer of the server to decide whether the documents should be sent as one or two messages.

Fig. 7 Underspecification using alt

Semantically, the alt operator is captured by the operator \(\cup\) for inner union. Let \((p_1, n_1)\) and \((p_2, n_2)\) be the interaction obligations corresponding to the two operands of the alt. The semantics of the overall construct is then captured by \((p_1, n_1) \cup (p_2, n_2) = (p_1 \cup p_2, n_1 \cup n_2)\).

The sendDoc diagram has no negative traces since neither of the operands of the alt specify negative behavior. On the other hand, sendDoc has three positive traces, one from the first operand of the alt and two from the second, since the interleaving of \(ld_2\) and \(?d_1\) is left open.

Notice that a sequence diagram may be specified such that there is an overlap between the positive and the negative traces of the interaction obligation \((p, n)\), i.e., \(p \cap n \neq \emptyset\). In that case, the positive traces that are also negative are treated as negative. The valid or legal traces of a specification are represented by the set \(p \setminus n\) and are referred to as the implementable traces of the interaction obligation.

3.2 Sequence Diagrams with xalt

In STAIRS, the alt operator captures non-deterministic choice in the meaning of underspecification, as already exemplified in Fig. 7. To specify inherent non-determinism, which may be understood as non-determinism the system is required to possess, we may use the xalt operator in STAIRS. This requires, however, a richer semantics. Instead of a single interaction obligation, we now need a set of interaction obligations to capture the meaning of a sequence diagram.

Fig. 8 illustrates the use of the xalt operator. The specification says that the user can either retrieve data from or store data on the server. Importantly, the server must offer both alternatives.

Semantically, the xalt operator corresponds to ordinary union of the sets of interaction obligations for its operands. Since neither of the operands of this particular example have occurrences of xalt, they are singleton sets. The xalt construct then denotes the union of the two sets, i.e., \(\{(\{lr, ?r, ?l, ?d\}, \emptyset)\}, \{(\{s, ?s, ?ls, ?o\}, \emptyset)\}\).

The semantics of sequence diagrams is defined by a function \(\llbracket\) that for any given diagram \(d\) yields a set \(\llbracket d \rrbracket = \{(p_1, n_1), \ldots, (p_m, n_m)\}\) of interaction obligations. It is formally defined as follows.

**Definition 1** Semantics of sequence diagrams.

\[
\begin{align*}
\llbracket e \rrbracket & \equiv \{(\{e\}, \emptyset)\} \text{ for any } e \in E \\
\llbracket \text{seq}\{d_1, d_2\} \rrbracket & \equiv \llbracket o_1 \supseteq o_2 \mid o_1 \llbracket d_1 \rrbracket \land o_2 \llbracket d_2 \rrbracket \\
\llbracket \text{neg}\{d\} \rrbracket & \equiv \{(\{\emptyset\}, p \cup n) \mid (p, n) \llbracket d \rrbracket \} \\
\llbracket \text{alt}\{d_1, d_2\} \rrbracket & \equiv \llbracket o_1 \cup o_2 \mid o_1 \llbracket d_1 \rrbracket \land o_2 \llbracket d_2 \rrbracket \\
\llbracket \text{xalt}\{d_1, d_2\} \rrbracket & \equiv \llbracket d_1 \rrbracket \cup \llbracket d_2 \rrbracket
\end{align*}
\]

3.3 Refinement

Refinement means adding information to a specification such that the specification becomes more complete and detailed. This may be achieved by reducing underspecification through redefining previously positive traces as negative or by redefining previously inconclusive traces as negative.

In STAIRS [9], there are also more general notions of refinement in which, for example, inconclusive traces may be redefined as positive. These more general notions of refinement are mainly of relevance in very specific phases of system development and are not considered in this paper.
Refinement facilitates analysis through abstraction, but for analysis at an abstract level to be meaningful, the analysis results must be preserved under refinement. The refinement relation should also be transitive so as to support a stepwise development process ensuring that the final, most complete specification is a valid refinement of the initial specification. Modularity of the development process should also be supported by monotonicity of the composition operators with respect to refinement.

By \((p, n) \rightarrow (p', n')\), we denote that the interaction obligation \((p', n')\) is a refinement of the interaction obligation \((p, n)\). Formally, refinement of interaction obligations is defined as follows.

**Definition 2** Refinement of interaction obligations.

\((p, n) \rightarrow (p', n') \equiv (n \subseteq n') \land (p \subseteq p' \cup n') \land (p' \subseteq p)\)

This means that in a refinement step, both inconclusive and positive traces can be redefined as negative. Once negative, a trace remains negative, while a positive trace cannot become inconclusive.

The variation over positive traces of an interaction obligation may reflect variations over the design choices under consideration. At later phases of the development, when the system in question is better understood and the specification is more complete, design choices are made by redefining some of the previously positive traces as negative. The negative traces of an interaction obligation explicitly state that these traces are held as invalid or illegal alternatives for fulfilling the interaction obligations. Hence, when design alternatives become more explicit through refinement, the negative traces remain negative.

For sequence diagrams \(d\) and \(d'\), refinement is defined as follows.

**Definition 3** Refinement of sequence diagrams.

\[
[d] \rightarrow [d'] \equiv \forall o \in [d] : \exists o' \in [d'] : o \rightarrow o' \land \\
\forall o' \in [d'] : \exists o \in [d] : o \rightarrow o'
\]

By this definition, all interaction obligations of a specification must be represented by an interaction obligation in the refined specification, and adding interaction obligations that do not refine an interaction obligation of the abstract level is not allowed. This ensures that the inherent non-determinism expressed by the variation over interaction obligations is persistent through refinement.

Since trace-set properties are expressed as properties of sets of interaction obligations in STAIRS, the refinement relation ensures that trace-set properties are preserved under refinement by the preservation of interaction obligations.

The following theorem implies that the refinement relation supports a stepwise development process.

**Theorem 1** The refinement relation is transitive and reflexive. Formally, for all sequence diagrams \(d_1, d_2,\) and \(d_3,\)

\[
[d_1] \rightarrow [d_2] \land [d_2] \rightarrow [d_3] \Rightarrow [d_1] \rightarrow [d_3]
\]

\[
[d_1] \rightarrow [d_1]
\]

For a formal proof of the theorem, see [34].

The following set of monotonicity results is established in [34]. These are important since they imply that the different parts of a sequence diagram can be refined separately.

**Theorem 2** If \([d_1] \rightarrow [d_1']\) and \([d_2] \rightarrow [d_2']\), then the following holds.

\[
[d_1][d_2] \rightarrow [d_1'][d_2']
\]

\[
[neg[d_1]][d_2] \rightarrow [neg[d_1']][d_2]
\]

\[
[seq[d_1,d_2]] \rightarrow [seq[d_1',d_2']]
\]

\[
[alt[d_1,d_2]] \rightarrow [alt[d_1',d_2']]
\]

\[
[xalt[d_1,d_2]] \rightarrow [xalt[d_1',d_2']]
\]

A sequence diagram is specified by composing sub-diagrams using the various operators. The monotonicity result, along with reflexivity, ensures that refinement of the sub-diagrams in isolation results in a valid refinement of the diagram as a whole. Moreover, the transitivity result means that each separate part of the sequence diagram can be subject to any number of refinement steps in order to refine the whole specification.

4 Secure Information Flow Properties

Secure information flow properties impose requirements on information flow between different security domains. The underlying idea is that an observer residing in one security domain (call it low) shall not, based on its observations, be able to deduce whether behavior associated with another security domain (call it high) has, or has not occurred. Such requirements are referred to as secure information flow properties (or information flow properties for short).

**Example 1** Suppose we require that the low-level user (\(\text{:UserL}\)) of Fig. 9 should not be able to deduce that the high-level user (\(\text{:UserH}\)) has done something. In the worst case, the low-level user has complete knowledge of the specification \(DS\). If the low-level user makes an observation, he can use this knowledge to construct the set of all the positive traces that are compatible with that observation (the low-level equivalence set). He can conclude that one of the traces in this set has occurred, but not which one.

With respect to Fig. 9, the low-level user can make three observations: \(\langle s, ?_o \rangle, \langle ?, \rangle, \) and \(\langle s, ?c \rangle\). If the observation \(\langle s, ?_o \rangle\) is made, he can conclude that either the trace describing the top-most scenario of Fig. 9 has occurred or that one of the traces describing the lower-most scenario has occurred. The trace of the top-most
scenario does not contain any events of the high-level user. Therefore, the low-level user cannot conclude with certainty that the high-level user has done something when the observation \( \langle \text{ls}, \text{?o} \rangle \) is made. However, if the low-level user makes the observation \( \langle \text{ls}, \text{?e} \rangle \), he can conclude with certainty that the high-level user has done something. This is because the low-level user receives an error message only when storing a document at the same time as the high-level user is attempting to retrieve the same document. Similarly, when observation \( \langle \text{s} \rangle \) is made, the low-level user knows for sure that the high-level user has done something. Hence, \( DS \) is not secure w.r.t. our requirement.

4.1 Capturing Information Flow Properties

The above example indicates that we need two ingredients to define precisely what is meant by an information flow property:

- a notion of high-level behavior that should always be a possibility, which we represent by a predicate \( H \in \mathcal{T} \rightarrow \text{Bool} \) on traces; (this predicate may, as in the previous example, characterize all traces in which a high-level user has not done something)
- a set \( L \subseteq \mathcal{E} \) of low-level events that can be observed by the low-level user

Intuitively, a system adheres to an information flow property \((L, H)\) if for each trace \( t \) of the system, there is a system trace \( t' \) that satisfies \( H \) and is low-level equivalent to \( t \). We say that two traces are low-level equivalent iff they cannot be distinguished by the observations made by the low-level user.

Let \( S \) be a system specification and suppose the low-level user has complete knowledge of \( S \). The low-level user may then, for each (low-level) observation he can make of the system, use \( S \) to construct the set that contains traces that are compatible with that observation. He will know that one of the traces in this set has occurred, but not which one. However, if each trace that is compatible with a given (low-level) observation fulfills \( H \), he can conclude with certainty that a high-level behavior has occurred.

This is illustrated in Fig. 10, where the outer rectangle depicts the set of all traces \( \mathcal{T} \). The ellipses depict the sets of positive traces in \( S \) that are compatible with the observations the low-level user can make of the system. The low-level observations are described by the traces \( l_1, l_2, \) and \( l_3 \) of low-level events, so for all positive traces \( t \) in \( S \), \( t \) is compatible with one of the observations \( l_1, l_2, \) or \( l_3 \). The dashed rectangle \( H \) depicts the high-level behavior. In this case, the set of traces compatible with the low-level observation \( l_2 \) is a subset of the high-level behavior. Hence, if the low-level user observes \( l_2 \), he knows with certainty that high-level behavior has taken place.

Likewise, if \( H \) is disjoint from a set of traces compatible with a low-level observation, the low-level user may with certainty deduce that no behavior characterized by \( H \) has taken place. In this case, information flows from the high-level domain to the low-level domain since the low-level user knows that the high-level behavior has not been conducted.

This is illustrated in Fig. 10 by the low-level observation \( l_3 \). Since all positive traces of \( S \) that are compatible with \( l_3 \) are not in \( H \), the low-level user knows for sure that no high-level behavior has taken place by observing \( l_3 \).

To prevent the low-level user from deducing that high-level behavior has occurred or that high-level behavior has not occurred, there must for each low-level observation exist both a compatible trace within \( H \) and a compatible trace within its complement \( \overline{H} \).

This is illustrated in Fig. 11. For example, by observing the trace \( l_1 \) of low-level events, the low-level user cannot deduce whether or not high-level behavior has occurred since \( l_1 \) is compatible with both possibilities.
Example 2 The requirement of Example 1 is captured by the security property known as non-inference \([28]\). In order to formalize non-inference, we distinguish between the set \(L\) of events that, as already mentioned, can be observed by the low-level user and the set \(C\) of events that should be considered confidential, i.e., \(L \cap C = \emptyset\).

Non-inference may now be captured by the information flow property \((L, H)\), where the high level predicate is specified as follows.

\[
H(t) \overset{\text{def}}{=} C \odot t = \langle \rangle
\]

For a trace \(t\) and a set of events \(E\), the filtering operation \(E \odot t\) yields the trace obtained from \(t\) by filtering away every event not contained in \(E\).

In some cases, it is necessary to distinguish between several high-level behaviors in order to capture adequate information flow properties. Since the low-level user has full knowledge of the specification \(S\), he may, for example, identify a subset \(H'\) of the high-level behavior characterized by \(H\) in which a specific high-level behavior is conducted, e.g., document retrieval. In Fig. 12, the set \(H'\) is within the intersection of \(H\) and the traces compatible with \(l_2\). By observing \(l_1\) or \(l_3\), the low-level user can deduce with certainty that high-level behavior that fulfills \(H'\) has not occurred. In order to prevent this, we may specify a set of predicates, each characterizing a high-level behavior, and require that each of them are compatible with all low-level observations.

As a further illustration of the need to operate with a set of high-level behaviors, suppose we want to prevent a low-level user from deducing which login password has been chosen by a high-level user. The set \(H\) that characterizes all traces in which the high-level user has not chosen a password can be used to prevent the low-level user from deducing that some password has been selected, but we may be satisfied by preventing the low-level user from deducing which password has been selected. In that case, we need a predicate \(H\) for each password that can be selected.

This is illustrated in Fig. 13, where we distinguish between three different high-level behaviors, \(H_1, H_2\) and \(H_3\). The low-level user can make three different observations, and in each case may deduce that a high-level behavior has been conducted. However, the low-level user cannot deduce which one, since each of the observations \(l_1, l_2,\) and \(l_3\) is compatible with all three high-level behaviors.

In the general case, we specify the high-level behaviors by a predicate on traces parameterized over the set of all traces, \(H \in T \rightarrow (T \rightarrow \text{Bool})\). Hence, the set of traces \(t'\) that fulfill the predicate \(H(t)\) characterizes a specific high-level behavior dependent on \(t\).

In some cases we may wish to impose requirements on information flow only if certain restrictions hold. We may, for example, wish to only prevent the low-level user from deducing that a particular document has been stored on a server if the low-level user has not been granted privileges to deduce this. Or we may wish to only prevent the low-level user from deducing that confidential events have not occurred if the occurrence is not influenced by the behavior of the low-level user. The latter assumption, for example, is made in the definition of the so-called perfect security property \([44]\).

Formally, the restriction is also specified by a predicate on traces parameterized over the set of all traces, \(R \in T \rightarrow (T \rightarrow \text{Bool})\). The idea is that if the low-level user makes the low-level observation corresponding to a trace \(t\), we are then only concerned about the high-level
behaviors of relevance for \( t \), which are those generated from the traces that fulfill the restriction predicate \( R(t) \).

This is illustrated in Fig. 14. The solid ellipse depicts the set that contains traces that are compatible with the low-level observation of the trace \( t_1 \), and the dashed rectangle represents those traces that are not high-level behaviors with respect to \( t_2 \). By observing the trace \( t_1 \), the low-level user can deduce with certainty that high-level behavior as characterized by \( H(t_2) \) has not occurred.

Suppose, now, that the trace \( t_2 \) for some reason or another is not relevant when trace \( t_1 \) has occurred. Then the predicate \( R(t_1) \) can be used to express this by defining \( R(t_1)(t_2) \Rightarrow \text{False} \). Hence, with respect to the observation \( t_1 \), the trace \( t_2 \) is not held as relevant for the security requirement in question. With this weakening of the requirement, the situation depicted in Fig. 14 illustrates a secure case.

![Fig. 14 Security requirement with restriction](image)

To summarize, inspired by [22], we specify a basic information flow property by a triple \((R, L, H)\) of a restriction predicate \( R \in T \rightarrow (T \rightarrow \text{Bool}) \), a set of low-level events \( L \subseteq E \), and a high-level predicate \( H \in T \rightarrow (T \rightarrow \text{Bool}) \). Intuitively, a system adheres to a basic information flow property \((R, L, H)\) if, for any trace \( t_1 \), the system is secure with respect to any high-level predicate \( H(t_2) \) such that \( R(t_1)(t_2) \), i.e., any high-level predicate \( H(t_2) \) of relevance for \( t_1 \). We specify an information flow property by a set \( P \) of such triples, each specifying a basic information flow property.

**Example 3** In the following, we demonstrate how to specify an information flow property using the approach introduced above.

Assume we want to capture the following property: the low-level user should not be able to deduce \((A)\) whether high-level behavior has occurred and \((B)\) whether a particular high-level behavior has not occurred.

We formalize the requirements \((A)\) and \((B)\) by the triples \((R_A, L, H_A)\) and \((R_B, L, H_B)\), respectively. Let \( L \subseteq E \) be the set of events that has the low-level user as transmitter or receiver, and \( C \subseteq E \) be the set of confidential events, i.e., the events that have the high-level user as transmitter or receiver. For simplicity’s sake, we assume that \( L \cup C = E \). The triple \((R_A, L, H_A)\), may then be specified as in Example 2 with \( R_A(t_1)(t_2) \Rightarrow \text{True} \) for all traces \( t_1, t_2 \).

Requirement \((B)\) asserts not only the occurrence of high-level events as confidential, but also the particular sequence of high-level events in a trace. This means that we must also take relative timing into consideration. The traces \( (l, c) \) and \((c, l)\) (where \( l \in L \) and \( c \in C \)), for example, should not be considered equal at the high-level since \( c \) occurs at different points in the two traces.

We specify \( H_B(t) \) in terms of a function \( h \in T \rightarrow T \) that yields the sequence of high-level events of a trace in which relative timing is reflected. \( h(t) \) is defined as the trace obtained from \( t \) by replacing each low-level event in \( t \) by a dummy event. The trace is truncated at the point in which the last high-level event occurs. Formally, the function \( h \) is defined by the following conditional equations.

\[
\begin{align*}
C \cap t = \emptyset \Rightarrow h(t) &= \emptyset \\
\forall e \in L \Rightarrow h(e \sim t) &= \langle e \rangle \smallsetminus h(t) \\
\forall e \not\in L \Rightarrow h(e \sim t) &= \langle e \rangle \smallsetminus h(t)
\end{align*}
\]

Let us say, for example, that the events occurring on the \( :UserH \) lifeline in Fig. 9 are high-level and that all other events are low-level. For the uppermost scenario, we then have \( h(\langle !s, ?s, !o, ?o, !d, ?d \rangle) = \emptyset \). The lowermost scenario is specified by a parallel composition of two sub-scenarios that yields all possible interleaveings as its semantics. The following are two examples of high-level sequences in which timing is considered.

\[
\begin{align*}
h(\langle !r, \langle !s, r, !d, ?s, ?d, ?o \rangle \rangle) &= \langle \langle r, \langle !s, r, r, ?, ?, d \rangle \rangle, \langle r, \langle !s, r, r, ?, ?, d \rangle \rangle \rangle \\
h(\langle !s, ?s, !r, !o, ?s, ?o, !d, ?d \rangle) &= \langle \langle r, \langle !s, r, r, ?, ?, d \rangle \rangle, \langle r, \langle !s, r, r, ?, ?, d \rangle \rangle \rangle
\end{align*}
\]

With the help of \( h, H_B \) may then be specified as follows.

\[
H_B(t)(t') \defn h(t) = h(t')
\]

The restriction is that we are only concerned about the high-level behaviors that are not influenced by the low-level user. This means that we are only interested in those traces \( t_2 \) for which there exists a trace \( t_1 \) with a common prefix with \( t_2 \) and whose first deviation from \( t_1 \) involves a high-level event. This restriction may be formalized as follows.

\[
R_B(t_1)(t_2) \defn \exists t \in T : \exists e \in C : t \subseteq t_1 \land t \smallsetminus \langle e \rangle \subseteq t_2 \land t \smallsetminus \langle e \rangle \not\subseteq t_1
\]

The expression \( t' \not\subseteq t \) yields true iff \( t' \) is a prefix of or equal to \( t \). If low-level behavior influences high-level behavior – for example, if low-level events are recorded at the high-level – the knowledge of this high-level behavior does not represent a security breach since it is information flow that is already allowed. In the specification of the predicate \( R_B \), these situations are ruled out.

\footnote{For a formal definition of the prefix relation \( \subseteq \) and its negation \( \not\subseteq \), see Appendix A.1.}
The triple \((R_B, L, H_B)\) requires that all observations the low-level user can make of the system are compatible with all possible sequences of high-level events in which relative timing is taken into consideration, provided the high-level behavior is not influenced by the low-level user.

### 4.2 Information Flow Property Adherence

So far we have explained what it means for a system to adhere to an information flow property. In the following, we formally define what it means for a specification that is preserved under refinement to adhere to an information flow property.

Adherence of a system specification \(S\) to an information flow property \(P\) means that the specification \(S\) satisfies the given property, i.e., \(S\) is secure w.r.t. the set of basic information flow properties. This is captured by the adherence relation \(\rightarrow_a\) and denoted \(P \rightarrow_a S\). In order to formally define this relation, we first define what it means that a system specification adheres to a basic information flow property \((R, L, H) \in P\), denoted \((R, L, H) \rightarrow_a S\). In the following, we let \(\hat{S}\) denote the set of all implementable traces of the specification \(S\), i.e.,

\[
\hat{S} = \bigcup_{(p,n) \in [S]} p \setminus n
\]

**Definition 4** Adherence to basic information flow property \((R, L, H)\) of system specification \(S\).

\[(R, L, H) \rightarrow_a S \iff \forall (p,n) \in \hat{S} : (p \setminus n) \cap [S] \neq \emptyset \iff \exists t_1, t_2 \in \hat{S} : R(t_1)(t_2) \Rightarrow \exists (p,n) \in [S] : L \sqcap t_1 = L \sqcap t_3 \land H(t_2)(t_3)
\]

Observe firstly that this definition captures the requirement that a basic information flow property applies only for high-level properties of relevance for \(t_1\), i.e., those \(H(t_2)\) such that the antecedent \(R(t_1)(t_2)\) is satisfied. Secondly, adherence to a basic information flow property is assured not by a single trace, but by an interaction obligation \((p,n)\). Since interaction obligations are preserved by refinement, so is adherence. In summary, \(R(t_1)\) identifies the set of high-level behaviors of relevance for \(t_1\). To make sure that each of these high-level behaviors of relevance are fulfilled by the specification and maintained through later refinements, there must exist for each behavior an interaction obligation whose implementable traces all fulfill the high-level behavior and are indistinguishable from \(t_1\) for the low-level user.

Adherence to an information flow property is now defined as follows.

**Definition 5** Adherence to an information flow property \(P\) of system specification \(S\).

\[P \rightarrow_a S \iff \forall (R, L, H) \in P : (R, L, H) \rightarrow_a S\]

**Example 4** Consider the system specification \(S\) depicted in Fig. 15 showing low-level users and high-level users interacting with a server. The referred diagram \(sendDoc\) is depicted in Fig. 7, where \(:User\) is substituted by \(:UserH\).

![Fig. 15 System specification](image-url)

Assume that the property of non-inference as expressed in Example 2 is imposed as a requirement. In this example, \(L\) (resp. \(C\)) is the set of events that has \(:UserL\) (resp. \(:UserH\)) as transmitter or receiver.

The semantics of the specification in Fig. 15 is given by two interaction obligations in which there are no negative traces. The interaction obligation of the upper operand of the \(\text{xalt}\) operator consists of the singleton set \(\{\langle s, ?, o, \omega \rangle\}\) of positive traces, where the low-level user observes the trace \(\langle s, \omega \rangle\). The interaction obligation of the lower operand consists of several positive traces, all of which have high-level events and are observed as \(\langle !s, ?o \rangle\) by the low-level user. Adherence is ensured by the former interaction obligation. Its only trace has no high-level events and is low-level equivalent to all positive traces in the specification.

Def. 4 demonstrates the STAIRS expressiveness to capture trace-set properties. Information flow property adherence requires the existence of an interaction obligation, the positive traces of which all comply with the high-level behavior. Adherence to an information flow property cannot be falsified by a single interaction obligation since compliance to the high-level behavior may be ensured by other interaction obligations of the specification.

### 4.3 Preservation of Information Flow Properties under Refinement

As illustrated in Fig. 1, where an information flow property is depicted to the left and refinements of system specifications are depicted to the right, the refinement relation preserves information flow property adherence. This result is captured by the following theorem.
Theorem 3  Preservation of information flow property adherence under refinement.

\[ P \rightarrow a S_1 \land S_1 \rightsquigarrow S_2 \Rightarrow P \rightarrow a S_2 \]

Proof Assume \( P \rightarrow a S_1 \) and \( S_1 \rightsquigarrow S_2 \). Prove \( P \rightarrow a S_2 \).

Let \( t_1 \) and \( t_2 \) be any elements of \( S_2 \) such that \( R(t_1)(t_2) \). By Def. 3 and the assumption that \( S_1 \rightsquigarrow S_2 \), \( t_1, t_2 \in S_1 \). Since \( R(t_1)(t_2) \) and \( P \rightarrow a S_1 \) by assumption, \( \exists (p, n) \in S_1 : \forall t \in (p \setminus n) : L \otimes t_1 = L \otimes t \land H(t_2)(t) \).

By Def. 3, \( \exists (p', n') \in S_2 : (p, n) \rightsquigarrow (p', n') \). By Def. 2, \( (p' \setminus n') \subseteq (p \setminus n) \), so \( \forall t' \in (p' \setminus n') : L \otimes t_1 = L \otimes t' \land H(t_2)(t') \), i.e., \( P \rightarrow a S_2 \) as desired. □

Importantly, by Theorem 3, if adherence is verified at an abstract level, this need not be verified again at the refined levels. Hence, STAIRS does not suffer from the so-called refinement paradox [15,16] in which trace-set properties are not preserved under refinement. In the following, we explain in more detail why this is so.

Underspecification arises when a term has several valid interpretations. The standard notion of refinement by underspecification [11] states that a system specification \( S' \) describing a set of traces \( T' \) is a refinement of a system specification \( S \) describing a set of traces \( T \) iff \( T' \subseteq T \).

Intuitively, there are at least as many implementations that satisfy \( S \) as there are implementations that satisfy \( S' \). In this sense, \( S' \) describes its set of implementations more accurately than \( S \), i.e., \( S' \) is equally or less abstract than \( S \). Adherence to secure information flow properties is, however, in general not preserved under this standard notion of refinement.

A system is secure with respect to an information flow property \( P \) if for all traces \( t_1 \) and \( t_2 \) such that \( R(t_1)(t_2) \), there is a non-empty set of traces fulfilling \( H(t_2) \). However, by defining refinement as set inclusion, there is no guarantee that a given refinement will keep any of the traces described by \( H(t_2) \) at the abstract level. Hence, information flow properties are in general not preserved by the standard notion of refinement.

Intuitively, the cause of the problem is that information flow properties depend on unpredictability. The strength of one’s password, for example, may be measured in terms of how hard it is for an attacker to guess the chosen password. The demand for the presence of traces fulfilling \( H(t_2) \) may be seen as a requirement of unpredictability, but traces that provide this unpredictability may be removed during refinement (if we use the standard notion). This motivates Def. 4 of adherence to a basic security predicate in which the distinction between underspecification and unpredictability is taken into consideration by requiring the existence of an interaction obligation whose non-negative traces all fulfill \( H(t_2) \). The STAIRS expressiveness to specify inherent non-determinism as variation over interaction obligations captures this. Furthermore, the preservation of this variation under refinement ensures preservation of information flow property adherence, as formalized by Theorem 3.

Example 5 We showed in Example 4 that the system specification \( S \) depicted in Fig. 15 adheres to the information flow property of non-inference. Assume, now, a specification \( S' \) in which \( sendDoc \) of Fig. 7 is replaced with \( sendDoc2 \) of Fig. 16.

The refuse operator is introduced in STAIRS as a variant of the UML neg operator. Semantically, the difference from neg is that rather than defining the singleton set \( \{ \} \) of the empty trace as positive, refuse yields the empty set \( \emptyset \) as the set of positive traces. Introducing the refuse operator is motivated by the fact that the neg operator is ambiguously defined in the UML standard. See [32] for further details on this issue.

In sendDoc2, the alternative of sending the two requested documents as separate messages, which is positive in sendDoc, is specified as negative. It is easy to see that sendDoc \( \rightsquigarrow \) sendDoc2. By the monotonicity results of Theorem 2, since sendDoc \( \rightsquigarrow \) sendDoc2, then also \( S \rightsquigarrow S' \).

![Fig. 16 Refined interaction](image)

5 Policies

During the last decade, policy based management of information systems has been subject to increased attention, and several frameworks (see e.g., [38]) have been introduced for the purpose of policy specification, analysis, and enforcement. At the same time, UML 2.1 has emerged as the de facto standard for the modeling and specification of information systems. However, the UML offers little specialized support for the specification of policies and for the analysis of policies in a UML setting.

In the following sections, we suggest how the UML in combination with STAIRS can be utilized for policy
specification. Furthermore, given a UML specification of a system for which a policy applies, we formally express what should be the relation between the policy specification and the system specification, i.e., we formally define the relation of policy adherence.

In Section 5.1, we introduce the syntax and semantics of our sequence diagram notation for policy specification. In Section 5.2, we define the notion of policy adherence for system specifications and show that policy adherence involves trace-set properties. In Section 5.3, preservation of adherence under refinement is addressed.

5.1 Specifying Policies

A policy is defined as a set of rules governing the choices in the behavior of a system [37]. A key feature of policies is that they “define choices in behavior in terms of the conditions under which predefined operations or actions can be invoked rather than changing the functionality of the actual operations themselves” [38]. This means that the potential behavior of the system generally spans wider than that which is prescribed by the policy, i.e., the system can potentially violate the policy. A policy can therefore be understood as a set of normative rules.

In our approach, each rule is classified as either a permission, an obligation, or a prohibition. This classification is based on standard deontic logic [26], and several of the existing approaches to policy specification have language constructs of such a deontic type [1,17,37,40,43]. This categorization is furthermore used in the ISO/IEC standardized reference model for open, distributed processing [14].

In [39], we evaluate UML sequence diagrams as a notation for policy specification. It is argued that although the notation is to a large extent sufficiently expressive, it is not optimal for policy specification. The reason for this lies heavily in the fact that the UML has no constructs customized for expressing the deontic modalities. In this section, we extend the sequence diagram syntax with constructs suitable for policy specification, and we define their semantics.

In our notation, a policy rule is specified as a sequence diagram that consists of two parts: a triggering scenario and a deontic expression. The notation is exemplified with the permission rule read in Fig. 17. The diagram captures the rule, stating that by the event of a valid login, the user is permitted to retrieve documents from the server, provided the user is a registered user.

The triggering scenario is captured with the keyword trigger, and the scenario must be fulfilled for the rule to apply. This is exemplified by the diagram userRegistered followed by the message login(id) in Fig. 17. Observe that in userRegistered, there are two alternative ways for a user to be registered. Either the user registers on personal initiative by sending a message with a chosen user id, or the server creates an account and sends the user id to the user. With respect to the rule in question, the two alternatives are considered equivalent.

The specification of a triggering scenario can be understood as a variant of a triggering event that is often used in policy specification [37]. By specifying the policy trigger as an interaction, the circumstances under which a rule applies can be expressed both as a simple message or as a more complex scenario that must be fulfilled.

Notice that we allow the trigger to be omitted in the specification of a policy rule. In that case, the rule applies under all circumstances and is referred to as a standing rule.

The behavior that is constrained by the rule is annotated with one of the keywords permission, obligation, or prohibition, where the keyword indicates the modality of the rule. In the example, the interaction describing document retrieval from the server by the user is specified as a permission. Two alternative ways of sending the document from the server to the user are specified as depicted in the referred diagram sendDoc of Fig. 7. Since the rule in Fig. 17 is a permission, the behavior specified in the body is defined as being permitted. By definition of a policy, a policy specification is given as a set of rules, each rule specified in the form shown in Fig. 17.

The proposed extension of the UML 2.1 sequence diagram notation to capture policies is modest and conservative, so people who are familiar with the UML should be able to understand and use the extended notation. Furthermore, all the constructs that are available in the UML for specification of sequence diagrams can be freely used in the specification of a policy rule.

In this paper, we do not consider refinement of policy specifications, only refinement of system specifications expressed in STAIRS. We assume that the policy specification is the final, complete specifications to be enforced. In that case, all behavior that is not explicitly specified as positive is negative, i.e., there are no incon-
clusive traces. Since \((p \cup n) = T\) in the cases of complete specifications, it suffices to refer only to the set of positive traces \(p\) in the semantics of the policy specifications.

The semantics of a specification of a policy rule is captured by a tuple \((dm, A, B)\), where \(dm \in \{pe, ob, pr\}\) denotes the deontic modality, \(A\) denotes the traces representing the triggering scenario, and \(B\) denotes the traces representing the behavior. For the special case of a standing policy rule, i.e., a rule in which the trigger is omitted and that applies under all circumstances, the semantics \(A\) of the triggering scenario is the set of all traces \(T\). Since a policy specification is a set of policy rule specifications, the semantics of a policy specification is given by a set \(P = \{r_1, \ldots, r_m\}\), where each \(r_i\) is a tuple capturing the semantics of a policy rule specification.

5.2 Policy Adherence

In order to fully understand the meaning of a policy specification, it is crucial to understand what it means that a given system specification \(S\) adheres to a policy specification \(P\). In the following, we define this adherence relation, denoted \(P \rightarrow_a S\). We assume a STAIRS system specification with sequence diagrams and let \([S]\) denote the STAIRS semantics of the specification, i.e., \([S]\) is a set of interaction obligations.

If a policy rule specification is triggered at some point in a system run, the rule imposes a constraint on the continuation of the execution. An obligation requires that for all continuations, the behavior specified by the rule must be fulfilled, whereas a prohibition requires that none of the continuations may fulfill the behavior. Hence, adherence to obligations and prohibitions are trace properties since they can be falsified on a single trace. A permission, on the other hand, requires that the sub-trace is an element of the positive traces set of \(S\) and that the super-trace is an element of \(S\) is a set of interaction obligations.

As we shall see, these trace properties and trace-set properties are generalized to properties of interaction obligations and sets of interaction obligations, respectively, in STAIRS system specifications.

Generally, a policy specification refers only to the system behavior of relevance for the purpose of the policy, which means that there may be scenarios described in the system specification that are not mentioned in the policy specification. For the same reason, scenarios of the system specification may be only partially described in the policy specification. Consider, for example, the diagram \textit{userRegistered} of the trigger in Fig. 17. For a system specification to fulfill this scenario, there must exist a scenario in the system specification in which \textit{userRegistered} is a sub-scenario. Assume that the diagram in Fig. 18 is the system specification of the registration of a new user. The user signs up with an id, typically a chosen username and password, after which the server tests the id, e.g., to ensure that the username is unique and that the password conforms to some rules. Since the trace of events representing \textit{register} contains the trace of events representing one of the alternatives specified in \textit{userRegistered}, \textit{userRegistered} is a sub-scenario of \textit{register}, i.e., \textit{register} represents a fulfillment of \textit{userRegistered}.

The fulfillment of one scenario by another is captured by the sub-trace relation \(<\). The expression \(t_1 \prec t_2\) evaluates to true iff \(t_1\) is contained in \(t_2\). We say that \(t_1\) is a sub-trace of \(t_2\) and that \(t_2\) is a super-trace of \(t_1\). We have, for example, that

\[\langle l, ?, r, ?, o, ?, o \rangle \prec \langle l, ?, r, l, c, ?, c, ?, o, ?, o \rangle\]

where the sub-trace is an element of the positive traces of \textit{userRegistered}, and the super-trace is an element of the positive traces of \textit{register}.\textsuperscript{3}

We generalize \(<\) to handle trace sets in the first argument. For a trace \(t\) to fulfill a scenario represented by a trace set \(T\), there must exist a trace \(t' \in T\) such that \(t' \prec t\), denoted \(T \prec t\). Formally,

\[T \prec t \iff \exists t' \in T : t' \prec t\]

The negated relation \(\prec\) is defined by the following.

\[t_1 \not\prec t_2 \iff -(t_1 \prec t_2)\]

\[T \not\prec t \iff \neg \exists t' \in T : t' \prec t\]

For a policy rule \((dm, A, B)\), we may now define what it means that an interaction obligation triggers the rule. Intuitively, an interaction obligation \((p, n)\) triggers the given rule if some of the implementable traces \(p \setminus n\) fulfill the triggering scenario \(A\). This is formally captured by the following definition.

\textbf{Definition 6} The rule \((dm, A, B)\) is triggered by the interaction obligation \((p, n)\) iff

\[\exists t \in (p \setminus n) : A \prec t\]

Notice that since the semantics of the triggering scenario is \(A = T\) for standing rules, these rules are trivially triggered by all interaction obligations with a non-empty set of implementable traces.

\textsuperscript{3} For a formal definition of the sub-trace relation \(<\), see Appendix A.3.
Example 6 The triggering of a rule is exemplified by the interaction overview diagram in Fig. 19 depicting a system specification. Semantically, the system specification $S$ defines two interaction obligations:

$$\{o_1\} = \{\text{register} \supset \text{login} \supset \text{retrieve} \supset \text{logout}\}$$

$$\{o_2\} = \{\text{register} \supset \text{login} \supset \text{store} \supset \text{logout}\}$$

As explained above, register fulfills userRegistered of the rule read in Fig. 17. It is furthermore easy to see that login of Fig. 20 fulfills the remainder of the triggering scenario, so the composition $\{\text{register}\} \supset \{\text{login}\}$ fulfills the triggering scenario of the permission rule. Hence, both $o_1$ and $o_2$ trigger the permission rule.

If a system specification $S$ adheres to a permission rule $(pe, A, B)$, whenever the rule is triggered by an interaction obligation $(p, n) \in \[S\]$, there must exist an interaction obligation $(p', n') \in \[S\]$ that triggers the same rule and also fulfills the permitted behavior. On the other hand, if $S$ adheres to an obligation rule $(ob, A, B)$, and the rule is triggered by an interaction obligation $(p, n) \in \[S\]$, $(p, n)$ must fulfill the behavior. Finally, if $S$ adheres to a prohibition rule $(pr, A, B)$, and the rule is triggered by an interaction obligation $(p, n) \in \[S\]$, $(p, n)$ must not fulfill the behavior.

As an example, consider again the permission rule read in Fig. 17 and the system specification given by the interaction overview diagram in Fig. 19 that triggers the rule. It is easy to see that interaction obligation $o_1$ fulfills the permitted behavior by the diagram retrieve in Fig. 20. As before, fulfillment of one scenario by another is defined by the sub-trace relation.

Formally, policy adherence is defined as follows.

Definition 7 Adherence to policy rule of system specification $S$.

$$- (pe, A, B) \rightarrow_a S \overset{def}{=} \exists (p, n) \in \[S\] : \exists t \in (p_i \setminus n_i) : A \circ t \Rightarrow \exists (p_i, n_j) \in \[S\] : \forall t' \in (p_j \setminus n_j) : (A \supset B) \circ t'$$

$$- (ob, A, B) \rightarrow_a S \overset{def}{=} \forall (p, n) \in \[S\] : \forall t \in (p \setminus n) : A \circ t \Rightarrow (A \supset B) \circ t$$

$$- (pr, A, B) \rightarrow_a S \overset{def}{=} \forall (p, n) \in \[S\] : \forall t \in (p \setminus n) : A \circ t \Rightarrow (A \supset B) \not\circ t$$

With these definitions of adherence to policy rules, we define adherence to a policy specification as follows.

Definition 8 Adherence to policy specification $P$ of system specification $S$.

$$P \rightarrow_a S \overset{def}{=} \forall r \in P : r \rightarrow_a S$$

5.3 Preservation of Policy Adherence under Refinement

The property of adherence preserving refinement is desirable, since it implies that if adherence is verified at an abstract level, adherence is guaranteed for all refinements and need not be verified again.

In order to formally prove that policy adherence is preserved under refinement, we must prove preservation of adherence under refinement for each kind of policy rule.

Theorem 4 Preservation of policy adherence under refinement.

(a) $(pe, A, B) \rightarrow_a S_1 \wedge S_1 \rightarrow S_2 \Rightarrow (pe, A, B) \rightarrow_a S_2$

(b) $(ob, A, B) \rightarrow_a S_1 \wedge S_1 \rightarrow S_2 \Rightarrow (ob, A, B) \rightarrow_a S_2$

(c) $(pr, A, B) \rightarrow_a S_1 \wedge S_1 \rightarrow S_2 \Rightarrow (pr, A, B) \rightarrow_a S_2$

Proof To prove (a), assume $(pe, A, B) \rightarrow_a S_1$ and $S_1 \rightarrow S_2$. Prove $(pe, A, B) \rightarrow_a S_2$.

Let $(p_i, n_i)$ be any element of $[S_2]$ such that there exists $t \in (p_i \setminus n_i)$ and $A \circ t$. The permission rule is then triggered by $(p_i, n_i)$, and we need to prove that there is an element $(p_j, n_j) \in [S_2]$ that triggers the same rule and that also fulfills the permitted behavior.
By Def. 3, \( \exists (p'_j, n'_j) \in [S_1] : (p'_j, n'_j) \leadsto (p_j, n_j) \). By Def. 2, \( p_i \subseteq p'_j \) and \( n'_j \subseteq n_i \). Hence, \( t \in (p'_j \setminus n'_j) \) and \( (p'_j, n'_j) \) triggers the permission rule. Since \((p, A, B) \leadsto_a S_1\) by assumption, \( \exists (p_j', n_j') \in [S_1] : \forall t' \in (p_j' \setminus n_j') : (A \succeq B) \not\subseteq t' \).

By Def. 3, \( \exists (p_j, n_j) \in [S_2] : (p_j', n_j') \leadsto (p_j, n_j) \). By Def. 2, \( p_j \subseteq p_j' \) and \( n_j' \subseteq n_j \), i.e., \( (p_j \setminus n_j) \subseteq (p_j' \setminus n_j') \). Hence, \( \forall t' \in (p_j \setminus n_j) : (A \succeq B) \not\subseteq t' \) as desired.

To prove (b), assume \((ob, A, B) \leadsto_a S_1\) and \( S_1 \leadsto S_2 \). Prove \((ob, A, B) \leadsto_a S_2\).

Let \((p, n)\) be any element of \([S_2]\) such that there exists \( t \in (p \setminus n) \) and \( A \not\subseteq t \). The obligation rule is then triggered by \((p, n)\), and we need to prove that \((A \succeq B) \not\subseteq t \).

By Def. 3, \( \exists (p'_j, n'_j) \in [S_1] : (p'_j, n'_j) \leadsto (p, n) \). By Def. 2, \( p \subseteq p'_j \) and \( n'_j \subseteq n \). Hence, \( t \in (p'_j \setminus n'_j) \), and \((p'_j, n'_j)\) triggers the obligation rule. By assumption of adherence, \((A \succeq B) \not\subseteq t \) as desired.

To prove (c), assume \((pr, A, B) \leadsto_a S_1\) and \( S_1 \leadsto S_2 \). Prove \((pr, A, B) \leadsto_a S_2\).

Let \((p, n)\) be any element of \([S_2]\) such that there exists \( t \in (p \setminus n) \) and \( A \not\subseteq t \). The prohibition rule is then triggered by \((p, n)\), and we need to prove that \((A \succeq B) \not\subseteq t \).

By Def. 3, \( \exists (p'_j, n'_j) \in [S_1] : (p'_j, n'_j) \leadsto (p, n) \). By Def. 2, \( p \subseteq p'_j \) and \( n'_j \subseteq n \). Hence, \( t \in (p'_j \setminus n'_j) \), and \((p'_j, n'_j)\) triggers the prohibition rule. By assumption of adherence, \((A \succeq B) \not\subseteq t \) as desired. \(\square\)

**Example 7** To illustrate the usefulness of Theorem 4, assume a specification \( S' \) that is identical to the specification of \( S \) depicted in Fig. 19, except that the diagram sendDoc of Fig. 7 is replaced with the diagram sendDoc2 of Fig. 16.

As explained in Example 5, since sendDoc \(\not\rightarrow\) sendDoc2, then also \( S \not\rightarrow S' \). In this step of refinement, underspecification is reduced by moving from \( S \) to \( S' \), but since \( P \rightarrow_a S \) and \( S \rightarrow S' \), we have \( P \rightarrow_a S' \) by Theorem 4.

The property of refinement that every interaction obligation \((p, n)\) at the abstract level is represented at the refined level by an interaction obligation \((p', n')\) such that \((p, n) \rightarrow (p', n')\) is crucial for the validity of Theorem 4: if an interaction obligation \((p, n)\) ensures adherence to a permission rule at the abstract level, then all refinements of \((p, n)\) will ensure the same adherence. For obligations and prohibitions, adherence is preserved since no interaction obligations can be added at the refined level that are not represented at the abstract level, thus ensuring that a policy breach is not introduced.

**6 Related Work**

It is well known that trace-properties such as safety and liveness are preserved under the classical notion of refinement defined as reverse set inclusion. Trace-set properties, however, are generally not preserved under this notion of refinement, which has given rise to the so-called refinement paradox [16]. As explained above, the STAIRS approach avoids this paradox by offering a semantics that is sufficiently rich to distinguish underspecification from inherent non-determinism. In the following, we relate STAIRS to other approaches to system specification and development with interactions.

The family of approaches we consider are those that have emerged from the ITU recommendation message sequence chart (MSC) [13], a notation most of which has been adopted and extended by the UML 2.1 sequence diagram notation. ITU has also provided a formal operational semantics of MSCs [12] based on work by Mauw and Reniers [23, 24].

An MSC describes a scenario by showing how components or instances interact in a system by the exchange of messages. Messages in MSCs, are as for UML sequence diagrams, ordered by weak sequencing, which yields a partial ordering of events. Several operators are defined for the composition of MSCs, such as weak sequencing, parallel composition, and alternative executions. The explicit specification of scenarios as negative or forbidden is, however, not supported.

More important in our context is that the distinction between underspecification and inherent non-determinism is beyond the standard MSC language and its semantics [12, 13]. Furthermore, there is no notion of refinement defined for MSCs. This means that there is no support in MSCs for the capture of trace-set properties and their preservation under a stepwise development process.

In [18], Katoen and Lambert define a compositional denotational semantics for MSCs based on the notion of partial-order multi-sets. This denotational semantics complements the standardized operational semantics, and does not aim to introduce new expressiveness. Hence, there is no distinction between underspecification and inherent non-determinism. Refinement is also not addressed in [18].

In the work by Krüger [19], a variant of MSCs is given a formal semantics and provided a formal notion of refinement. Four different interpretations of MSCs are proposed; namely, existential, universal, exact, and negative. The existential interpretation requires the fulfillment of the MSC in question by at least one system execution; the universal interpretation requires the fulfillment of the MSC in all executions; the exact interpretation is a strengthening of the universal interpretation by explicitly prohibiting behaviors other than the ones specified by the MSC in question; the negative interpretation requires that no execution is allowed to fulfill the MSC. Three notions of refinement are defined; namely, property, message and, structural. Property refinement corresponds to the classical notion of refinement as reverse set inclusion, i.e., the removal of underspecification by reducing the possible behavior of the system; message
refinement is to substitute an interaction sequence for a specific message; structural refinement means to decompose an instance, or lifeline, with a set of instances.

The interesting interpretation in our context is the existential scenario as it may capture trace-set properties. As shown in [19], a system that fulfills an MSC specification under the universal, exact, or negative interpretation also fulfills specifications that are property refinements of the initial specification. This is, however, not the case for the existential interpretation, which means that trace-set properties are not preserved under refinement. Broy [3] proposes an interpretation of MSCs that differs significantly from the standardized approach. Rather than understanding an MSC as a description of the entire system as a whole, MSCs are in his work perceived as descriptions of properties of each system component (instance or lifeline in MSC and UML terms, respectively). In the semantic model, MSCs are described in terms of logical propositions that characterize stream-processing functions. A stream is basically a sequence of events, and a component is described by a function from input streams to output streams. If an MSC is not specified for a given input stream, the behavior is inconclusive. This interpretation deviates from the standard interpretation of both MSCs and UML sequence diagrams.

The paper discusses how MSCs should be utilized in the development process and is supported by a formal notion of refinement. Refinement is defined in the style of Focus [4] on the basis of the input/output function of a single component and with respect to a given input stream. Basically, a component is a refinement of another component if for a given input stream, the set of output streams generated by the former is a subset of the set of output streams generated by the latter. Hence, the concrete component fulfills all the requirements that are fulfilled by the abstract component. The approach in [3] does not address trace-set properties.

Uchitel et al. [42] present a technique for generating Modal Transition Systems (MTSs) from a specification given as a combination of system properties and scenario specifications. System properties, such as safety or liveness, are universal as they impose requirements on all system runs, and are in [42] specified in Fluent Linear Temporal Scenario. Scenario specifications are existential as they provide examples of intended system behavior, and are specified in the form of MSCs. An MTS is a behavior model with the expressiveness to distinguish between required, possible, and proscribed behavior. The method for generating MTSs from properties and scenarios ensures that all system behaviors satisfy the properties while the system may potentially fulfill all the scenarios. Furthermore, both composition and refinement of behavior models preserve the original properties and scenarios. With this approach, MSCs can be utilized to capture trace-set properties and preserve these under refinement. However, as opposed to e.g., STAIRS, the approach in [42] is not a pure interaction based development process, and the use of MSCs is restricted to the existential interpretation. Moreover, reasoning about system behavior and refinement is based on the MTS behavior models and cannot be easily conducted at the syntactic level.

Grosu and Smolka [6] address the problem of equipping UML 2.0 sequence diagrams with a formal semantics that allows compositional refinement. In their approach, positive and negative interactions are interpreted as liveness and safety properties, respectively. This may obviously be a useful interpretation of sequence diagrams, but is much stronger than the traditional interpretation in which sequence diagrams are used to illustrate example runs. Refinement is defined as the traditional reverse trace set inclusion and is compositional. There is, however, no support for the preservation of trace-set properties under this notion of refinement.

Störrle [41] defines a trace semantics for UML 2.0 interactions where a sequence diagram is captured by a pair of a set of positive traces and a set of negative traces. The approach provides no support for distinguishing between underspecification and inherent non-determinism. Rather, positive traces are interpreted as necessary, i.e., must be possible in an implementation, whereas negative traces are interpreted as forbidden, i.e., must not be possible in an implementation. A notion of refinement is introduced in which previously inconclusive traces may be redefined as positive or negative. With this approach, there is no guarantee that adherence to trace-set properties is preserved under refinement.

Live sequence charts (LSCs) [5,8] are an extension of MSCs that particularly address the issue of expressing liveness properties. LSCs support the specification of two types of diagrams; namely, existential and universal. An existential diagram describes an example scenario that must be satisfied by at least one system run, whereas a universal diagram describes a scenario that must be satisfied by all system runs. Universal charts can furthermore be specified as conditional scenarios by the specification of a prechart that, if successfully executed by a system run, requires the fulfillment of the scenario described in the chart body. The universal/existential distinction is a distinction between mandatory and provisional behavior, respectively. Such a distinction is also made between elements of a single LSC by characterizing these as hot or cold, where a hot element is mandatory and a cold element is provisional. LSCs furthermore have the expressiveness to specify forbidden scenarios by placing a hot condition that evaluates to false immediately after the relevant scenario. The condition construct of LSCs and MSCs corresponds to UML 2.1 state invariants and is a condition that must evaluate to true when a given state is active. If and when the system fulfills the given scenario,
it is then required to satisfy the false condition, which is impossible.

LSCs seem to have the expressiveness to ensure adherence to trace-set properties by the use of existential diagrams; obviously, falsification of system satisfaction of requirements expressed by an existential diagram cannot be done on a single trace. However, a system development in LSCs is intended to undergo a shift from an existential view in the initial phases to a universal view in later stages as knowledge of the system evolves. Such a development process with LSCs will generally not preserve trace-set properties. Moving from an existential view to a universal view can be understood as a form of refinement, but LSCs are not supported by a formal notion of refinement.

The semantics of LSCs is that of partial ordering of events defined for MSCs [13]. Importantly, however, the semantics of LSCs defines the beginning of precharts and the beginning of main charts as synchronization points, meaning that all lifelines enter the prechart simultaneously and that the main chart is entered only after all lifelines have completed their respective activities in the prechart. This yields a strong sequencing between the prechart and main chart, a property of LSCs that may seem unfortunate in some situations, specifically for distributed systems where system entities behave locally and interact with other entities asynchronously. A further disadvantage with respect to system development and analysis with LSCs is the lack of composition constructs for combining LSCs. A specification is instead given as a set of LSCs, and a system satisfies a specification by satisfying each of its individual LSCs.

Modal sequence diagrams (MSDs) [7] are defined as a UML 2.0 profile. The notation is an extension of the UML sequence diagram notation based on the universal/existential distinction of LSCs. The main motivation for the development of the MSD language are the problematic definitions of the assert and negate constructs of UML sequence diagrams. The authors observe that the UML 2.0 specification is contradictory in the definition of these constructs, and also claim that the UML trace semantics of valid and invalid traces is inadequate for properly supporting an effective use of the constructs.

The semantics for MSDs is basically the same as for LSCs. The main difference is that the LSC prechart construct is left out. Instead, a more general approach is adopted in which cold fragments inside universal diagrams serve the purpose of a prechart. A cold fragment is not required to be satisfied by all runs, but if it is satisfied, it requires the satisfaction of the subsequent hot fragment.

With respect to capturing trace-set properties and preserving these under refinement, the situation is the same as for LSCs. Existential diagrams may be utilized to ensure adherence to trace-set properties. However, there is no formal notion of refinement guaranteeing preservation of adherence. The existential view is understood as a form of underspecification suitable for the early phases. A gradual shift from existential to universal diagrams reduces the underspecification and restricts the set of system models that satisfy the specification. It is in [7] indicated that existential diagrams should be kept in the specification as the development process evolves and that universal diagrams are gradually added. With such an approach, trace-set properties might be preserved.

Triggered message sequence charts (TMSCs) [36] is an approach in the family of MSCs that is related to STAIRS in several ways. The development of TMSCs is motivated by the fact that MSCs do not have the expressiveness to define conditional scenarios, i.e., that one interaction (the triggering scenario) requires the execution of another (the action scenario), that MSCs do not formally define a notion of refinement, and that MSCs lack structuring mechanisms that properly define ways of grouping scenarios together.

The triggering scenarios of TMSCs are closely related to the precharts of LSCs. An important semantic difference, however, is that whereas LSCs are synchronized at the beginning of precharts and main charts, TMSCs are based on weak sequencing in the spirit of MSCs.

TMSCs have composition operators for sequential composition, recursion (similar to loop), and parallel composition. The most interesting composition operators in the context of this paper, however, are delayed choice and internal choice. Both operators take a set of diagrams as operands and define a choice between these. Internal choice is an operator in the spirit of the STAIRS alt operator and defines underspecification. An implementation that can execute only one of the operands is a correct implementation. Delayed choice, on the other hand, is an operator somewhat related to the STAIRS alt operator; a correct implementation must provide all the choices defined by the specification. However, as opposed to inherent non-determinism captured by the xalt operator, a delayed choice is not made until a given execution forces it to be made.

It is observed in [36] that the simple trace set semantics of MSCs does not have the expressiveness to distinguish between optional and required behavior, which means that a heterogeneous mix of these in the specification is not supported. Such a mix is supported in STAIRS; syntactically by the alt operator for underspecification and the xalt operator for inherent non-determinism, semantically by the set of interaction obligations.

The semantics of TMSCs is defined in terms of so-called acceptance trees. Acceptance trees record the traces that are defined by a specification, but also distinguish between required and optional behavior. Importantly, the semantics of TMSCs supports a notion of refinement that preserves the required behavior and gives freedom with respect to optional behavior.

The authors stress that refinement should preserve properties such as safety and liveness. These properties are trace properties, and the issue of capturing trace-
set properties and preserving these under refinement is not discussed. However, by the semantics of the delayed choice operator and the fact that this type of choice is preserved under refinement, TMSCs seem to have the expressiveness to capture and preserve trace-set properties.

Like the refinement relations of STAIRS, refinement of TMSCs is compositional, i.e., composition of specifications is monotonic with respect to refinement.

An important difference between STAIRS and TMSCs is that the latter do not support the specification of negative behavior. This also means that with TMSCs there is no notion of inconclusive behavior; a system specification defines a set of valid traces, and all other traces are invalid. The refinement relation is based on set inclusion, so refinement basically means that behavior that was previously defined as positive is redefined as negative. TMSCs allow the specification of partial scenarios in the sense that interactions may continue beyond what is specified in a given TMSC. This means that diagrams may be incomplete and that the traces are extensible through refinement. Semantically, partial scenarios mean that what is left unspecified by a scenario imposes no constraints on behavior, i.e., that all behavior beyond what is specified is legal.

Distinguishing between positive, negative, and inconclusive behavior as in STAIRS and UML 2.1 provides useful expressiveness and is, in our opinion, a more intuitive interpretation of partial or incomplete scenarios than what is provided for TMSCs. The lack of a precise definition in the UML 2.1 specification of what is the meaning of a valid trace motivated the work on MSDs. For both LSCs and MSDs, the precise meaning of a specification is given by defining what it means that a system model satisfies a specification. In STAIRS, this is correspondingly defined by formalizing the relation of compliance [34]. However, in the process of system development, it is as important to understand the meaning of the distinction between valid and invalid behavior when moving from one specification to a more concrete specification by refinement. This is precisely defined in STAIRS by the formal notion of refinement.

We conclude this section by relating our customized UML sequence diagram notation for policy specification presented in Section 5 to other work. The new features of this notation with respect to both UML sequence diagrams and MSCs are basically the triggering construct and the deontic modalities. Triggering scenarios for MSCs are supported in the work by Krüger, and also for LSCs and TMSCs. In the former two, the main motivation for specifying triggering scenarios is that they facilitate the capturing of liveness properties. In contrast to our approach, the composition of the triggering scenario and the triggered scenario is that of strong rather than weak sequencing in both Krüger’s work and for LSCs. Furthermore, Krüger proposes no constructs for explicitly characterizing scenarios as obligated, permitted, or prohibited, but rather operates with different interpretations of MSC specifications. In LSCs, scenarios are existential, universal, or forbidden, which corresponds closely to the permission, obligation, and prohibition modalities, respectively. There is, however, no explicit construct for specifying the latter. Instead, a hot false condition must be placed immediately after the relevant scenario. As for TMSCs, there is no support for distinguishing between obligated, permitted, and negative behavior; all traces are characterized as either valid or invalid.

7 Conclusions

The expressiveness of STAIRS allowing inherent nondeterminism to be distinguished from underspecification facilitates the specification of trace-set properties and the preservation of such properties under refinement. This paper demonstrates the potential of STAIRS in this respect within the areas of information flow security and policy specification.

STAIRS formalizes the trace semantics that is only informally described in the UML standard. STAIRS furthermore offers a rich set of refinement relations [34] of which only one (restricted limited) has been addressed in this paper. The refinement relations are supported by theoretical results indicating their suitability to formalize modular, incremental system development.

STAIRS also offers an operational semantics that has been proved to be sound and complete with respect to the denotational semantics [21]. The Escalator tool [20] makes use of this semantics to offer automatic verification and testing of refinements.

Furthermore, there are extensions of STAIRS to deal with (hard) real-time [10], as well as probability [30] and soft real-time [29].

Acknowledgements The research on which this paper reports has been partly funded by the Research Council of Norway through the projects ENFORCE (164382/V30) and DIGIT (180052/S10), and partly by the European Commission through the S3MS project (Contract no. 27004) under the IST Sixth Framework Programme. We are grateful to Mass Soldal Lund, Atle Refsdal, and Ragnhild Kobro Runde for valuable comments and suggestions.

References


The following operations on sequences are adapted from \cite{Stoerle99}. The prefix relation on sequences, \( \preceq \), is formally defined as follows.

\[ s_1 \preceq s_2 \iff \exists s \in E^\omega : s_1 \preceq s = s_2 \]

The complementary relation is defined by the following.

\[ s_1 \nsubseteq s_2 \iff \neg(s_1 \subseteq s_2) \]

The truncation operator

\[ \downarrow_i \in E^\omega \times \mathbb{N} \cup \{\infty\} \to E^\omega \]

is used to truncate a sequence at a given length.

\[ s|_j = \begin{cases} s' & \text{if } 0 \leq j \leq \#s, \text{where } \#s' = j \wedge s' \subseteq s \\ s & \text{if } j > \#s \end{cases} \]

\( \mathcal{P}(E) \) denotes the set of all subsets of \( E \). The filtering operator

\[ \odot_\prec \in \mathcal{P}(E) \times E^\omega \to E^\omega \]

is used to filter away elements. \( A \odot s \) denotes the subtrace of \( s \) obtained by removing elements of \( s \) that are not in \( A \). For a finite sequence \( s \), this operator is completely defined by the following conditional equations.

\[
\begin{align*}
A \odot \langle \rangle &= \langle \rangle \\
e \in A &\Rightarrow A \odot (e \prec s) = (e) \prec (A \odot s) \\
e \notin A &\Rightarrow A \odot (e \prec s) = A \odot s
\end{align*}
\]

For an infinite sequence \( s \), we need one additional equation.

\[ \forall n \in \mathbb{N} : s[n] \notin A \Rightarrow A \odot s = \langle \rangle \]

The filtering operator \( \odot \) is defined for pairs of sequences:

\[ \odot_\prec \in \mathcal{P}(E \times E) \times (E^\omega \times E^\omega) \to (E^\omega \times E^\omega) \]

In order to formally define this operator, we first generalize some of the above operators on sequences to pairs of sequences.

\[
\begin{align*}
\#(s_1, s_2) &= \min\{\#s_1, \#s_2\} \\
(s_1, s_2)[n] &= (s_1[n], s_2[n]) \\
(s_1, s_2) \prec (s_1', s_2') &= (s_1 \prec s_1', s_2 \prec s_2') \\
(s_1, s_2)|_j &= (s_1|_j, s_2|_j)
\end{align*}
\]

Furthermore, for elements \( e_1, e_2 \in E \), \((e_1, e_2)\) denotes \((\langle e_1 \rangle, \langle e_2 \rangle)\).

For a pair of sequences \( c = (s_1, s_2) \), the filtering operator \( \odot \) is now defined by the following conditional equations.

\[
\begin{align*}
B \odot c &= B \odot (c[\#c]) \\
B \odot (\langle \rangle, \langle \rangle) &= (\langle \rangle, \langle \rangle) \\
f \in B &\Rightarrow B \odot ((f) \prec c) = (f) \prec B \odot c \\
f \notin B &\Rightarrow B \odot ((f) \prec c) = B \odot c \\
\forall n < \#c + 1 : c[n] \notin B &\Rightarrow B \odot c = (\langle \rangle, \langle \rangle)
\end{align*}
\]
A.2 Weak Sequencing

For traces $t_1$ and $t_2$, which are sequences of events, weak sequencing is defined by the function

$$\preceq : \mathcal{T} \times \mathcal{T} \to \mathcal{P}(\mathcal{T})$$

Formally, weak sequencing is defined as follows.

$$t_1 \preceq t_2 \overset{\text{def}}{=} \{ t \in \mathcal{T} \mid \forall l \in \mathcal{L} : e.l \in t = e.l \in t_1 \dashv e.l \in t_2 \}$$

where $e.l$ denotes the set of events that may take place on the lifeline $l$. Formally,

$$e.l \overset{\text{def}}{=} \{ (k, (\text{co, tr, re})) \in \mathcal{E} \mid \begin{align*} & (k = ! \land tr = l) \lor (k = ? \land re = l) \end{align*} \}$$

A.3 Sub-trace Relation

The sub-trace relation $\preceq$ on sequences,

$$\preceq : \mathcal{E}^\omega \times \mathcal{E}^\omega \to \mathbb{B}$$

is formally defined as follows.

$$s_1 \preceq s_2 \overset{\text{def}}{=} \exists s \in \{1, 2\}^\infty : \pi_2((\{1\} \times \mathcal{E}) \oplus (s, s_2)) = s_1$$

$\pi_2$ is a projection operator returning the second element of a pair. The infinite sequence $s$ in the definition can be understood as an oracle that determines which of the events in $s_2$ are filtered away.
Chapter 13

A Method for Model-Driven Information Flow Security

Fredrik Seehusen\textsuperscript{1,2} and Ketil Stølen\textsuperscript{1,2}
\textsuperscript{1} SINTEF ICT, Norway
\{Fredrik.Seehusen, Ketil.Stolen\}@sintef.no
\textsuperscript{2} Department of Informatics, University of Oslo, Norway
2009

Abstract
We present a method for software development in which information flow security is taken into consideration from start to finish. Initially, the user of the method (i.e., a software developer) specifies the system architecture and selects a set of security requirements (in the form of secure information flow properties) that the system must adhere to. The user then specifies each component of the system architecture using UML inspired state machines, and refines/transforms these (abstract) state machines into concrete state machines. It is shown that if the abstract specification adheres to the security requirements, then so does the concrete one provided that certain conditions are satisfied.

1 Introduction
Security incidents occur on a daily basis within many companies. In CSI/FBI Computer Crime and Security Survey for 2005 [9], 74\% of the companies reported security incidents. Despite the importance of security, careful engineering of security into overall design is often neglected and security features are typically built into an application in an ad-hoc manner or are only integrated during the final phases of system development [22].

Model-Driven Security (MDS) [3] advocates the opposite. MDS aims to raise the level of abstraction in design and development of secure systems by supporting (1) a model-driven development process in which security is taken into account from start to finish, (2) a clear separation of abstract, platform independent models (PIMs) and refined, platform specific models (PSMs), and (3) adherence preserving transformations between PIMs and PSMs.

The central idea of MDS is that systems can be specified and shown to be in adherence with security requirements at different levels of abstraction. Abstraction is believed to simplify analysis, facilitate reuse of designs and early discovery of design flaws. At each level of abstraction, we distinguish between a system specification and a set of security requirements the system specification must adhere to. When we have established that a system specification adheres to the security requirements at a given level, this relationship should also hold
at the next level so that the invested effort to establish adherence is not wasted. Hence, we would like adherence to be preserved under transformation. Adding details to a specification may of course require some additional analysis. However, it should not be necessary to recheck the adherence relationship already established at the more abstract level.

The MDS framework is illustrated in Fig.1. Here a platform independent model together with its security requirements are transformed into several platform specific models with associated security requirements.

Already published approaches to MDS include [2, 3, 5, 8, 10, 14, 22, 37]. Although interesting, they are in most cases of a rather informal nature; the semantics of the languages used (at abstract or/and concrete levels) are not sufficiently precise to allow for rigorous reasoning at more than one level of abstraction. Moreover, some of the approaches ([2, 3, 8, 22]) consider transformation of security requirements only, and not transformation of system specifications. These approaches allow adherence checking only at the lowest level of abstraction. Others ([5, 8, 10, 37]) do not clearly characterize what it means for a system to adhere to a security requirement. Instead, security is described in terms of a security mechanism.

Security is often defined as the preservation of confidentiality, integrity, and availability [15]. In this report, we, however, focus on security in the more narrow sense of secure information flow properties (see e.g., [4, 11, 25, 27, 30, 31, 35, 38]) which provide an elegant way of specifying confidentiality as well as integrity requirements [26].

The notion of transformation is closely related to refinement. That is, refinement is the (possibly manual) process of making an abstract specification more concrete, whereas transformation typically is a special case of refinement in which this process is automatic. There are several kinds of refinement. One of the refinement notions considered in this report, is refinement w.r.t. under-specification, i.e., the process of removing alternative design choices that are equivalent in the sense that it suffices for an implementation to provide only of them. In the classical literature, this kind of refinement is often referred to as behavioral refinement or property refinement [6]. It has long been recog-
nized that secure information flow properties are not preserved under standard definitions of refinement w.r.t. underspecification [16]. We avoid this problem by defining refinement in a semantic framework which is more expressive than conventional frameworks. In our method, system specifications are written in a state machine notation inspired by UML. But contrary to UML state machines, we have two constructs of choice: one describing alternative design choices, and one describing choices which should be provided by the system. The distinction between the two kinds of choices is necessary in order to handle refinement of secure information flow properties [13, 17, 31].

Software systems are almost never built entirely from scratch, i.e., a set of already implemented operations is usually available from the operating systems, runtime environments, or from libraries of programming environments. To take this into account, our state machines are allowed to reference events that are already available in a predefined event library. These events are later substituted by their definitions by a so-called event transformation. The transformation may therefore be understood as a special case of what in the literature is known as action refinement [36]. We provide a formal characterization of the syntax and semantics of transformations that are induced by event libraries, and define general conditions under which transformations induced by event libraries preserve arbitrary secure information flow properties.

This report is structured as follows: Sect. 2 gives an overview of our method. In Sect. 3 - 9 the individual steps of our method are presented. Sect. 10 discusses related work, and Sect. 11 provides conclusions and directions of future work. In the appendix, we formally define state machines (App. A) and event transformations (App. B). A glossary of symbols is provided in App. C and proofs are presented in App. D.
2 Overview of method

The goal of our method for model-driven information flow security is to support the specification and development of secure software systems. As illustrated in Fig. 2, the method has seven main steps.

In step I, the user of our method (i.e., a software developer) specifies the system architecture using UML composite structures. The system architecture is an overview of the components of the system and their associated communication channels. The user then partitions the system into security domains by labeling the system architecture with security relevant annotations.

In step II, the user selects a set of secure information flow properties (typically from a library) that the specification must adhere to. The secure information flow properties of the library are assumed to be defined in our security property schema [33, 34] which ensures that the adherence to the properties is preserved when design decisions are resolved by refinement in later steps.

In step III, the user specifies each component of the system architecture using our UML inspired state machines. The state machine notation provides constructs for specifying both design choices and choices that must be offered by the components of the system. The specified state machines may reference events that are already provided in a predefined event library. In this step, new event specifications may also be uploaded to the event library.

In step IV, the user verifies that the state machine based specification adheres to the security properties selected in step II. There are many techniques and methods that can be used for this purpose. We do not go into details on these. However, we provide a precise characterization of what it means for a system specification to adhere to a secure information flow property. This characterization provides a formal foundation for adherence verification.

In step V, the user refines the specification by removing alternative design decisions until all design decisions are decided. The validity of adherence is guaranteed to be preserved under this kind of refinement.

In step VI, the state machine specification of step V is transformed into a more concrete specification by substituting the event references of the specification by their definitions in the event library. We give a formal characterization of the syntax and semantics of these so-called event transformations, and show that they satisfy some desirable properties.

In step VII, the user verifies that the event transformation of step VI preserves adherence to the security properties selected in step II. We present conditions that can be used to check that the transformation preserves adherence.

In the MDS terminology, the specification produced in step V may be seen as a platform independent model (PIM), whereas the state machine produced in step VII may be seen as a platform specific model (PSM). Note that although a fully general approach to MDS would consider the transformation of both (abstract) security properties and system specification into (concrete) security properties and system specifications, we only consider the transformation of the system specifications. The reason for this is that the security properties we consider are essentially parameterized w.r.t. to the different levels of abstraction. Hence, there is normally no need to transform the actual specification of the properties.
3 Step I: Specify and annotate the system architecture

In step I of our method, the user specifies the system architecture, i.e., an overview of the basic state machines that the system specification consists of and their associated communication channels. We will often refer to the architecture specification as a composite state machine. After the architecture has been specified, the user partitions the system into security domains by annotating the specification. In the following, we introduce a running example which will be used to explain this step and the subsequent steps of our method.

Consider a large software developing company that aims to develop a distributed system, the project management system (the PM system), in order to centralize all storage of software development projects. Software developers should be able to retrieve projects from a server to their local machines, edit or add files to the project, and upload any changes back to the server.

Currently, the company has no unified development method, and developers working on different projects are to a large degree given flexibility in the method they choose to adopt. The company wants to assess the different methods in order to recommend improvements, and possibly to introduce a unified development process. This task is assigned to a group of researches. The researchers are to pick a set of sample projects, and assess each project thoroughly with respect to progress, quality of code etc. For convenience, the PM system should be augmented slightly such that the researchers will be able to retrieve projects from the server over the Internet on a regular basis. This additional functionality is not of high priority, thus it will be implemented with little resources. It is not initially known whether the researchers should use the same client as the developers.

To make the assessment as realistic as possible, the developers should not know which projects the researchers have sampled for the assessment. As we will see later, this requirement may be enforced by selection an appropriate secure information flow property.

Fig. 3 shows the architecture specification of the PM system with one client acting on the behalf of the developer (ClientD), one client acting on the behalf of the researcher (ClientR), and one server (Server). The architecture specification is intended to give an overview of the components or basic state machines that the system specification consists of. As a graphical notation, we have chosen to use UML composite structures (see App. A for more details).

After the system architecture has been specified, the user partitions the architecture into security domains by annotating the components with the security domain they belong to. In our running example, developers are not allowed to find out which projects the researchers have sampled for the assessment. Thus
we let the developers belong to one security domain which we call the low level domain, and the researchers belong to another domain, which we call the high level domain. We require that information should not flow from the high level domain to the low level domain.

In Fig. 4, the domains that the developer and the researcher clients belong to are specified by the labels << Low >> and << High >>, respectively. For short, we denote the low level domain by $L$, and the high level domain by $H$. Semantically, we interpret a security domain by a set of events, where an event represents the transmission or the reception of a message. In the current example, this means that $L$ denotes all events that represent messages that can be sent to or from ClientD, whereas $H$ denotes all events that represent messages that can be sent to or from ClientR.

### 4 Step II: Select security properties

In step II, the user selects a set of secure information flow properties, referred to as security properties for short, that the system must adhere to. A security property defines what it means that information flows from one security domain to another.

The security requirement of the running example may be formalized by the security property non-inference [30] denoted $N_f$. This is one of the most well-known security properties of the literature. It treats all the behavior of the high level domain $H$ (the behavior of the researchers in this example) as confidential, and requires that the low level user (the developer) must not deduce that any event in $H$ has occurred.

We assume (1) that the developer may observe all events in $L$, i.e., its own communication with the server and (2) that the developer has complete knowledge of the system specification. Therefore, for each observation (i.e., a sequence of events in $L$) that the developer can make by interacting with the server, the developer can look at the specification and construct the so-called low level equivalence set of all traces (i.e., sequences of events describing system executions) that are compatible with that observation. The developer will know that one of the traces in this set has occurred, but not which one. However, if all the traces of this set include a high level event in $H$, then the developer can conclude with certainty that the researcher has done something. In this case, the specification would not be secure w.r.t. the NF property. Conversely, if there is one trace in the low level equivalence set which does not include a high level event, then the developer cannot conclude with certainty that the researcher has done something.

In general, the underlying idea of all secure information flow properties is to demand that each low level equivalence set must contain a trace which prevents
the low level user from deducing that some confidential behavior has (or has not) occurred.

The above discussion suggests that all security properties have two essential ingredients:

- a definition of low level equivalence;
- a definition of the high level behavior which should be regarded as confidential.

Formally, two traces \( s \) and \( t \) are low level equivalent, written \( s \sim_l t \), if they contain the same sequences of low level events, i.e.,

\[
\left| s \right|_L = \left| t \right|_L
\]

where \( \left| t \right|_L \) yields the trace obtained from \( t \) by removing all events not in the set of events \( L \). The definition of low level equivalence is the same for all security properties. The definition of confidential behavior, however, differs depending on the property. For the \( \text{Nf} \) property – which treats traces that contain any high level event in \( H \) as confidential – the set of confidential behavior \( C \) may be defined by the following predicate

\[
C(t) \overset{\text{def}}{=} t|_H \neq \langle \rangle
\]

where \( \langle \rangle \) denotes the empty trace. In other words, a trace is confidential if it is non-empty when all non high level events have been removed from it.

For a system \( S \), whose set of possible traces is denoted by \( \llbracket S \rrbracket \), to adhere to the \( \text{Nf} \) property, there must, for each low level observation that can be made from \( \llbracket S \rrbracket \), be a trace in \( \llbracket S \rrbracket \) which is not confidential (i.e., not included in \( C \)) and which is low level equivalent to the observation. Formally we have

\[
\text{Nf}(\llbracket S \rrbracket) \overset{\text{def}}{=} \forall t \in \llbracket S \rrbracket : \exists u \in \llbracket S \rrbracket : \neg C(u) \land u \sim_l t
\]

All security properties are of a similar form as (3). In fact, \( \text{Nf} \) may be expressed as an instance of a security predicate schema which is parameterized by a predicate which characterizes confidential behavior (see [34, 33]). We assume that all security properties that can be selected by the user in step II are instances of the security property schema.

5 Step III: Specify system components

In step III of our method, the user specifies each basic state machine of the architecture. It is assumed that a library of predefined event specifications is available to the user. These events can be referenced in the state machine specifications.

Fig. 5 shows the specification of the two clients ClientD and ClientR. ClientD receives an input document from its user (not shown), and passes that document on to the server by sending the message storeDoc. In the state machine notation, the black circle represents the initial state and the boxes with rounded edges represent standard states. Transitions are specified by arrows between states. Transitions may be labeled by action expressions of the form \( \text{nm.si}[bx]/ef \), where \( \text{nm.si} \) is an input event, \( [bx] \) (where \( bx \) is a boolean expression) is a
**STEP III: SPECIFY SYSTEM COMPONENTS**

Figure 5: Specification of the clients

`guard` and `ef` is an `effect`. Event triggers, guards, and effects are all optional in the expression.

Intuitively, if the current state of the state machine is `q`, and the state machine has a transition from `q` to state `q'` that is labeled by `nm.si(bx)/ef`, then the state machine will set its current state to `q'` if it receives signal `si` from the basic state machine whose name is `nm` and the guard `bx` is evaluated to true. The effect `ef` will be executed when the state machine moves to `q'`. There are two kinds of effects: output events and assignments. An output event is an expression of the form `nm.si` that represents the transmission of a signal `si` to the state machine whose name is `nm`. Assignments are denoted by expressions of the form `x = ex` where `x` is a variable and `ex` is a term built from common basic types such as booleans, integers, and strings, and common operations for these.

The graphical notation used for specifying state machines is essentially a subset of the UML state machine notation. See App. A for more details.

We assume that the message transmission protocol between the server and the client is provided by the event library. As specified in Fig. 5, the clients invoke the action `P1` (for protocol 1) in order to send messages to the server.

The client acting on the behalf of the researcher (`ClientR`) is similar to `ClientD`, except for two differences. First, the researcher retrieves documents from the server instead of storing them, and second `ClientR` has the choice of either using protocol `P1` or using protocol `P2`. This is indicated by the `alt`-construct which is used to specify design choices that are potential in the sense that one of the choice alternatives can be removed during refinement. In the current example, the design choice is specified because it is not initially known whether or not `ClientR` will use the same protocol as `ClientD`.

The specification of the server is shown in Fig. 6. Upon receiving the message `storeDoc` (resp. `getDoc`) from the client, it sends the message `storeDocument` (resp. `retrieveDoc`) to a data base (not shown) which stores (resp. retrieves) the document. The server, moves into a state in which it waits for an `ok` message from the data base. If the server receives any message from the clients while waiting for the data base, then these messages are simply dropped. As indicated by the `alt`-construct, the protocol used in order to transmit and receive messages to the researcher client is an open design choice.
The denotational semantics of standard state machines is usually given as the set of traces that is obtained by recording the events of all paths in the state machine when starting from the initial state. However, we distinguish between two kinds of choice, and interpret state machines as sets of trace sets called obligations. Intuitively, the traces of an obligation are equivalent in the sense that an implementation is only required to produce one of them. Thus, each trace of an obligation represent underspecification, or potential choices. However, an implementation is required to produce at least one trace in each obligation. Thus obligations represent explicit choices that have to be present in an implementation.

The obligations described by a basic state machine \( P \) is denoted by \( \llbracket P \rrbracket \) (see App. A.3 for a formal definition). Choices that are not potential will result in new obligations being created, whereas potential choices will result in obligations being collapsed. For instance, the semantics of \( \text{ClientD} \) is

\[
\llbracket \text{ClientD} \rrbracket = \{ \{ \} \}, \{ \{ u.i, !s.p1 \} \}, \{ \{ u.i, !s.p1, u.i, !s.p1 \} \}, \ldots
\]

Here a new obligation is created for each finite iteration of the loop in \( \text{ClientD} \). Each obligation consists of a single trace since there are no potential choices in \( \text{ClientD} \). Note that state machine names and signal names have been shortened, and that expressions of the form \(?nm.si\) and !\(nm.si\) represent input and output events, respectively.

The semantics of \( \text{ClientR} \) is given by

\[
\llbracket \text{ClientR} \rrbracket = \{ \{ \} \}, \{ \{ u.i, !s.p1 \}, \{ u.i, !s.p2 \} \}, \{ \{ u.i, !s.p1, u.i, !s.p1 \}, \{ u.i, !s.p2, u.i, !s.p2 \}, \ldots \}, \ldots
\]

Contrary to \( \text{ClientD} \), the obligations of \( \text{ClientR} \) are not singleton sets because \( \text{ClientR} \) contains potential choices. For instance, in obligation \( \{ \{ u.i, !s.p1 \}, \{ u.i, !s.p2 \} \} \), the traces \( \{ u.i, !s.p1 \} \) and \( \{ u.i, !s.p2 \} \) represent potential choices. Note that \( p1 \) and \( p2 \) for the time being are just syntactic suffixes of events. It is first when we make use of event libraries that they come into play.

We assume that basic state machines are autonomous, and therefore executed in parallel. Semantically, parallel composition is defined by interleaving traces. For instance, if the two traces \( \langle e_1, e_2 \rangle \) and \( \langle e'_1, e'_2 \rangle \) describe the execution of two different independent state machines, then the parallel composition of these traces, written \( \| \langle e_1, e_2 \rangle, \langle e'_1, e'_2 \rangle \), is given by

\[
\langle e_1, e_2, e'_1, e'_2 \rangle, \langle e'_1, e'_2, e_1, e_2 \rangle, \langle e_1, e'_1, e_2, e'_2 \rangle, \langle e'_1, e_1, e_2, e'_2 \rangle, \langle e_1, e'_1, e'_2, e_2 \rangle, \langle e'_1, e_1, e'_2, e_2 \rangle
\]

Each trace describes a possible execution of the two state machines that the traces \( \langle e_1, e_2 \rangle \) and \( \langle e'_1, e'_2 \rangle \) belong to. Because the state machines are independent and autonomous, all interleavings of the two traces are possible executions.

Basic state machines are usually not completely independent because a state machine that expects an input message from another state machine cannot proceed with execution before the message is received. Hence the execution of one state machine may be influenced by another. Traces that describe communication in which messages are sent before they are received are called well formed. We require that all traces in the semantics of a composite state machine to be well formed.
6 Step IV: Verify security adherence

In step IV, the user verifies that the specification adheres to the selected security property. Our method describes what it means for a specification to adhere to a security property, but leaves the choice of evaluation process to the user. The evaluation may be formal, e.g., by theorem proving or by model checking if feasible, or it may be by other means like testing if the specifications are too large for full verification. In the following, we argue only informally that our system specification adheres to the $N_f$ property.

In the definition of the $N_f$ property given in Sect. 4 (see (3)), we considered a system given as a trace set. Since we interpret specifications as sets of obligations, the definition needs to be revisited to take this into account. Instead of requiring that there is a trace $u$ which prevents the low level user from deducing that confidential behavior has occurred, we require that there is an obligation $\phi$ such that all its traces prevent the low level user from deducing that confidential behavior has occurred. This means that the obligations, as opposed to the individual traces, provide the unpredictability required by the security property. This is in line with the intuition behind the use of obligations. Formally, we have

$$NF([P]) \defeq \forall t \in [P] : \exists \phi \in [P] : \forall u \in \phi : \neg C(u) \land u \sim_t$$  \hspace{1cm} (4)
Note that $\hat{\Omega}$, where $\Omega$ is a set of obligations, collapses $\Omega$ into a set of traces.

Again, this definition of the $N_f$ property may be expressed as an instance of a security property schema which is defined for specifications that are interpreted as sets of obligations [33].

The PM system is secure w.r.t. the $N_f$ property because the developer can never be sure that the researcher has done something regardless of the low level observations he can make. There are two reasons for this. First it is always possible for the researchers to do nothing. Second, the developers cannot influence the behavior of the researcher and vice versa because neither the developers nor the researchers receive input from the server.

To see this, note that the semantics of ClientR includes the obligation $\{\emptyset\}$. By composing this obligation in parallel with the obligations of ClientD (which describe the observations that the developers can make), and Server, we obtain obligations whose traces do not contain any high level events, i.e., message transmission to or from ClientR. Therefore, for every observation that the developer can make, he can never be sure that the researchers have done something.

7 Step V: Refine system components

In Step V, the user refines the specification by removing alternative design choices until all choices are resolved. The security property selected in step II (defined by (Eq. (4))) is preserved under refinement because it is defined in a semantic model which distinguishes between potential choices and inherent choices that should be provided by a system.

Recall that in our running example, it was not initially known whether or not the researcher would use the same communication protocol as the developers for communicating with the server. Assume that it is decided that the researcher clients will use the same protocol as the developer clients. We then refine the specifications of the previous section by removing all the choice alternatives that involve protocol 2. Fig. 7 shows the resulting specification for the server. The same process has to be carried out for the researcher client.

We now make precise what is meant by refinement. Intuitively, obligations
are seen as providing alternative choices which must be provided by an implementation, whereas each trace in an obligation provides a design choice that may be removed under refinement. Formally, a state machine $Q$ is a refinement of a state machine $P$, written $P \rightsquigarrow Q$, iff

\[(\forall \phi \in \llbracket P \rrbracket : \exists \phi' \in \llbracket Q \rrbracket : \phi' \subseteq \phi) \land (\forall \phi' \in \llbracket Q \rrbracket : \exists \phi \in \llbracket P \rrbracket : \phi' \subseteq \phi)\]  

(5)

The reason why $\text{NF}$ property is preserved under this notion of refinement is that it is defined (see (4)) such that obligations (as opposed to traces which are used in most standard definitions) may be seen as providing the unpredictability required by the security property. Since choices provided by an obligation cannot be removed under refinement to an implementable specification, this means that the required unpredictability is preserved. Note that a specification whose semantics includes an empty set is not considered implementable. These kinds of specifications may therefore safely be ignored.

In [33] we show that any security property expressed in the security property schema is preserved under refinement.

8 Step VI: Event transformation

In this step VI, the specification (call it abstract) obtained in step V is transformed into a concrete one by replacing the events of the abstract specification by the specification of these events in the event library. At this stage, all design choices are assumed to be decided. In the following we therefore interpret a specification in the standard way as a trace set (see App. A for a formal definition).

An event library is a set of event specifications. An event specification consists of an event definition and a state machine pattern defining the behavior of the event. An event definition is a pair consisting of a kind (! or ?) and a signal pattern of the form $si(f_{p1}, \ldots, f_{pn})$ where $f_{p1}, \ldots, f_{pn}$ are formal parameters. A state machine pattern is a state machine that may contain signal patterns.

In the running example, it is assumed that the event specifications for $P1$ (which takes care of message transmission) are already specified in the library. The specifications of these events are shown in Fig. 8. The figure shows two specifications, one for the receive event (the signal name which is marked by ?) and one for the send event (the signal name which is marked by !). Both specifications have one formal parameter called $msg$.

The $P1$ protocol works as follows. After a message has been transmitted, the sender starts a timer, and waits for an acknowledgement (i.e., an $\text{ack}$ message) from the receiver. If an acknowledgement is not received within a certain time frame, then a timeout is triggered (the full details of this is not shown), and the sender retransmits the message. If an acknowledgement has not been received after the message has been transmitted 10 times, then the sender gives up. Note that the label $to$ in the output event specification (the topmost specification of Fig. 8) and the label $from$ (the lowermost specification of Fig. 8) are parameters that are bound to the transmitter and the receiver of the event being replaced, respectively. For instance, Fig. 9 shows the result of replacing the events containing $P1$ in the upper most specification of Fig. 5 by the event specification of Fig. 8. Here we see that the $to$ parameter has been bound to $\text{Server}$ which is the recipient of the message.
We let $T_{EL}$ be the function that takes a basic state machine $P$ and yields the state machine $T_{EL}(P)$ obtained from $P$ by replacing the events occurring in each action expression of $P$ by their definition in event library $EL$. For instance, if $EL$ is the event library of Fig 8, and $ClientD$ is the topmost specification of Fig. 5, then $T_{EL}(ClientD)$ yields the specification shown in Fig. 9.

To ensure that transformations preserve semantic equality, i.e., that two abstract specifications that are (semantically) equal are not transformed into two (semantically) different concrete specifications, we require all variables in event specifications to be local. To ensure this, we let $T_{EL}$ rename variables of the event specification such that no name clashes occur when events are being replaced by their definitions. For instance, if the topmost specification of Fig. 8 is applied in a context where the variable $c$ is used, then this variable is given a fresh name which is not used in the context.

In App. B, we show that any event transformation $T_{EL}$ preserve semantic equality, i.e.,

$$[P] = [Q] \implies [T_{EL}(P)] = [T_{EL}(Q)]$$

It is also shown that an event transformation is entirely characterized by its behavior on events. Technically, this means that the transformation is homomorphic w.r.t. the union and concatenation of trace sets.

The transformation $T_{EL}$ is defined for basic state machines. The transformation induced by an event library $EL$ which takes a composite state machine $P$ as input, is simply the function that applies $T_{EL}$ to all basic state machines that $P$ consists of.

The transformation of a composite state machine may produce concrete traces that have no abstract equivalent due to the granularity we get when events are substituted by state machines. We say that the image of a transformation $T_{EL}$, written $Im_{EL}$, is the set of all concrete traces that have an abstract equivalent. In App. B.2, we show that any event transformation $T_{EL}$ for composite state machines preserve semantic equality when restricting attention to the image, i.e.,

$$[P] = [Q] \implies ([T_{EL}^C(P)] \cap Im_{EL}) = ([T_{EL}^C(Q)] \cap Im_{EL})$$

It is also shown that the transformation $T_{EL}^C$ for composite state machines is homomorphic w.r.t. union of trace sets when restricting attention to the image.

9 Step VII: Verify adherence preservation

In step VII, the user verifies that the transformation performed in step VI is adherence preserving. Again, we do not consider any particular method of evaluation (e.g., model checking), instead we present the conditions that need to be evaluated.

We say that a transformation preserves a security property $SecP$ if the image of every abstract state machine $P$ that is secure w.r.t. $SecP$ is secure w.r.t. $SecP$, i.e.,

$$SecP([P]) \implies SecP([T_{EL}(P)] \cap Im_{EL})$$
We restrict attention to image since we cannot exploit the fact that the abstract specification is secure to ensure that concrete traces that do not have any abstract equivalent do not violate security. This means that additional security analysis may be needed at the concrete level to ensure that those traces that do not have any abstract equivalent do not violate security. However, it should not be necessary to recheck the adherence relationship already established at the more abstract level.

In App. B.3, we define a general condition under which an event transformation preserves a given security property. To obtain the specific condition under which the event library of step VII preserves the $N_f$ property, the general condition must be instantiated by the definition of the $N_f$ property. We then obtain the following condition:

(a) if an abstract trace contains no high level events, then no concrete traces it is transformed to can contain any high level events;

(b) if two abstract traces are low level equivalent, then no the concrete traces they are transformed to must also be low level equivalent.

Condition (a) is satisfied because the set of high level events is assumed to be all events that can occur in ClientR, and because event transformations cannot change the transmitters or receivers of events.
To check whether condition (b) is satisfied, we may check whether the parallel composition of all corresponding state machine patterns of event specifications in $EL$ (i.e., one send and one receive event) yields the same trace set regardless of context. In the current example, the event library of Fig. 8 satisfies this condition. For instance, the parallel composition of the two event specifications in Fig. 8 when substituted for event $!s.p_1$ (in ClientD of Fig. 5) and $?cd.p_1$ (in Server of Fig. 6), is

$$\{\langle !s.sd, ?cd.sd, !cd.a, ?s.a \rangle, \langle !s.sd, !s.sd, ?cd.sd, !cd.a, ?s.a \rangle, \ldots \}$$

Since none of the traces in this set contains a context dependent input event, all traces are well formed regardless of the context they appear in.

10 Related work

This report is related to previous work by the authors [33, 34]. In particular, the security property schema (referred to in Sect. 4) was introduced in [34, 33]. There it was also shown that all security properties of the schema are preserved under refinement. The main difference between this report and [33, 34], is that [33] does not consider transformations at all, and [34] is mostly semantics based; specifications are treated as trace sets, or sets of trace sets. Thus state machines are not considered at all, and a rigorous characterization of syntactic transformations and their interpretation is not given. The main contribution of this report is the definition of UML inspired state machines (Sect. 5 and App. A) and the definition of event transformations (Sect. 8 and App. B).

The state machine notation we use is inspired by UML. Several approaches have given UML state machines a formal semantics (see e.g., [1, 20, 29]). However, none of the works we are aware of distinguishes between explicit and potential nondeterminism.

The semantics of state machines that we propose is based on STAIRS [12], and our notion of refinement corresponds to so-called limited refinement in STAIRS. The main difference between our work and STAIRS is that STAIRS gives a semantics for UML sequence diagrams, whereas we consider state machines. Another difference is that the STAIRS denotational semantics with data [32] describes traces that will never be produced during execution because the values of the data states are allowed to change at any time between assignments. In our denotational semantics, we do not allow this. In this sense, our denotational semantics is more similar to the operational semantics of STAIRS [24].

Our work is related to two somewhat distinct areas of research: model-driven security (MDS) and information flow security. Already published approaches to MDS include [2, 3, 5, 8, 10, 14, 22, 37]. Although interesting, they differ from our work in that they are in most cases of a rather informal nature; the semantics of the languages used (at abstract or and concrete levels) are not sufficiently precise to allow for rigorous reasoning at more than one level of abstraction. Moreover, some of the approaches ([2, 3, 8, 22]) consider transformation of security requirements only, and not transformation of system specifications. These approaches allow adherence checking only at the lowest level of abstraction. Furthermore, others ([5, 8, 10, 37]) do not clearly characterize what it means for
a system to adhere to a security requirement. Instead, security is described in terms of a security mechanism.

Information flow security was originally introduced [35] as a generalization of the so-called non-interference property [11] from deterministic to nondeterministic systems. Since then, a number of different information flow properties (see e.g., [27, 30, 38]), as well general information flow frameworks (see e.g., [4, 25, 28, 31, 38]), have been proposed in various semantic models. The security schema considered in this report (App. B.3) is inspired by the framework of [25]. One of the main differences between the frameworks is that the distinction of potential and explicit choice is not made in [25].

Preservation of secure information flow properties under refinement was first considered in [16]. There it was shown that information flow properties are not in general preserved under a notion of refinement based on inverse trace set inclusion. It has later been observed (see e.g., [13, 17, 18, 31]) that the problem occurs in approaches that do not take the distinction of potential and explicit nondeterminism into account. All of the above citations consider a less general notion of refinement than we do.

We are not aware of any work which consider preservation of information flow properties under a syntactic notion of transformation.

11 Conclusions and future work

We have presented a method for model-driven information flow security. The method has 7 main steps that accommodate the integration of security into design artifacts above the code level. In particular, the method supports (1) specification of systems at different levels of abstraction; (2) rigorous adherence verification at different levels of abstraction; (3) adherence preserving refinement and transformation.

The method is based on UML inspired state machines that we have extended with a construct for specifying potential choice. These state machines, as well as event transformations from abstract to concrete state machines have been given formal syntax and semantics. We have shown that the event transformations satisfy properties that simplify the conditions under which adherence is preserved.

This report is related to previous work of the authors [34, 33]. However, state machines were not considered in [34, 33], and a formal definition of syntactic transformations and their interpretation was not given. The main contribution of this report is the definition of UML inspired state machines (Sect. 5 and App. A) and the definition of event transformations (Sect. 8 and App. B).

Many approaches to model-driven security exist, but all of them are of a rather informal nature. This makes it difficult to precisely define the meaning of adherence or preservation of adherence under refinement and transformation.

We are not aware of any work that address the preservation of information flow security properties under a syntactic notion of transformation.

In future work, we would like to develop syntactic type checking rules which can be used by a tool to automatically check preservation of adherence under transformations.
Acknowledgement  The research of this report has partly been funded by the Research Council of Norway project SECURIS (152839/220) and DIGIT (180052/S10), and partly by the EU through the S3MS (contract no. 27004) project.

References


A State Machines

In this section, we present the syntax and the semantics of our UML inspired state machines.


A.1 Syntax

A state machine is either basic in the sense that it does not consist of other state machines or composite in the sense that it does consist of other state machines. We first present the syntax of basic state machines, then we define the syntax of composite state machines.

A.1.1 Basic state machines

In this section, we first present the graphical syntax of basic UML inspired state machines. Then we define a textual syntax.

We make use of the following syntactic categories:

\[ ax \in \text{AExp} \] arithmetic expressions
\[ bx \in \text{BExp} \] boolean expressions
\[ sx \in \text{SExp} \] string expressions

Here \( ax, bx, \) and \( sx \) are syntactic variables of the three syntactic categories, respectively.

We let \( \text{Exp} \) denote the set of all arithmetic, boolean, and string expressions, and we let \( ex \) range over this set. Moreover, we assume that the following sets are given:

\[ x, y \in \text{Var} \] variables
\[ n \in \text{Num} \] numerals
\[ st \in \text{Str} \] strings

We let \( \text{Val} \) denote the set of all values, i.e., numerals, strings, and booleans (t or f).

As illustrated in Fig. 10, the constructs which are used for specifying state machines are initial state, simple state, accepting state, transition, action expression, and potential choice.

The state machine illustrated in Fig. 10 specifies a button that whenever pressed, displays the number of times it has been pressed to a user. The number is displayed by sending the signal \( \text{display}(num) \) or \( \text{displayHex}(num) \) (where \( num \) represents the number of times the button has been pressed) to the user. It is intended that \( \text{display}(num) \) is used for displaying the number in base 10, and that \( \text{displayHex}(num) \) is used for displaying the number in base 16.

Figure 10: Example of a basic state machine
A STATE MACHINES

Whether the button sends the signal \textit{display}(\textit{num}) or the signal \textit{displayHex}(\textit{num}) should be interpreted as a design choice as specified by the potential choice construct (\textit{alt}) attached to the branching transitions.

A state describes a period of time during the life of component. The three kinds of states, \textit{initial state}, \textit{simple state}, and \textit{accepting state}, are graphically represented by a black circle, a box with rounded edges, and a black circle encapsulated by another circle, respectively.

A \textit{transition} represents a move from one state to another. Transitions are labeled by \textit{action expressions} of the form

\[ nm.si[\textit{bx}]/ef \]

| Here the expression \textit{nm.si}, where \textit{nm} is a state machine name and \textit{si} is a signal, is called a \textit{trigger}. The expression \textit{[bx]} where \textit{bx} is a boolean expression is called a \textit{guard}, and \textit{ef} is called an \textit{effect}. Intuitively, the action should be understood as follows: when signal \textit{si} is received from a state machine with name \textit{nm} and the boolean expression \textit{bx} evaluates to true, then the effect \textit{ef} is executed. An effect is either an assignment or an output expression of the form \textit{nm.si} representing the transmission of signal \textit{si} to the state machine with name \textit{nm}.

An assignment is a pair \((x, ex)\) consisting of a variable \(x\) and an expression \(ex\). The set of all assignments is defined by

\[ \text{Assign} \overset{def}{=} \text{Var} \times \text{Exp} \]

In diagrams, assignments are written \(x = ex\) instead of \((x, ex)\).

We let \textbf{ActExp} denote the set of all action expressions and we let \textit{actx} range over this set. Triggers, guards, and effects are all optional parts of an action expression. We sometimes indicate the absence of these parts by \(\epsilon\), or we omit to write them all together. For instance, the expressions \(\epsilon[\textit{bx}]/\textit{ef}\) and \([\textit{bx}]/\textit{ef}\) are valid and equal action expressions without any trigger.

We let \textbf{Nm} denote the set of all state machine names and we let \textit{nm} range over this set. We represent names by strings, thus we have \(\textbf{Nm} \subseteq \text{Str}\). A signal is a tuple \((st, ex_1, \ldots, ex_n)\) where \(st\) denotes the signal name, and \(ex_1, \ldots, ex_n\) are the parameters of the signal. We usually write \(st(ex_1, ex_2, \ldots, ex_n)\) instead of \((st, ex_1, ex_2, \ldots, ex_n)\). Formally, the set of all signals is defined

\[ \text{SI} \overset{def}{=} \text{Str} \times \text{Exp}^* \]

The \textit{potential choice} construct (graphically represented by an \textit{alt}) may be attached to branching transitions in a state machine specification to specify alternative design choices.

The graphical notation used for specifying state machines is essentially a subset of the UML statechart notation. However, there are two differences:

- We allow the sender of an input signal to be specified on the transition labels. UML does not have any particular construct for specifying this.

- The potential choice construct is not part of the UML statechart notation.

In theory, the former difference could have been avoided by letting the name of the sender of an input signal be part of the name of the signal. However, by definition of event transformations, we would then have to specify one event
A STATE MACHINES

22

definition for each possible sender or receiver of a signal. This would be impractical. To see this, consider for instance the example of Sect. 8, where the state machine of Fig. 9 is obtained by replacing all signals whose name is \( P_1 \) in Fig. 5 according to the event definition of Fig. 8. Since the name of the sender or receiver of signals is not part of the signal name, we only have to specify two event definitions: one for the transmission of \( P_1 \) and one for the reception of \( P_1 \). However, if the sender or receiver names had been part of the signal name, then we would have to specify one event definition for each possible sender or receiver. This is clearly not a practical solution.

Note that we do not consider our manner of specifying transmission of signals in state machines a deviation from the UML standard. The reason for this is that the standard does not impose any particular restriction on the kind of effects that can be specified or the language they are specified in.

Having defined the graphical syntax of basic state machines, we now move on to define the textual syntax of basic state machines.

**Definition A.1 (Basic state machine expression)** The set of all basic state machine expressions \( P \) is defined by the following grammar

\[
P :: \text{act} | P^* | P.P | P + P | P|P
\]

The base cases implies that any action (\( \text{act} \)) is a basic state machine expression. Any other state machine expression is constructed from the basic ones through the application of the operators for iteration (\( P^* \)), sequential composition (\( P.P \)), standard choice (\( P + P \)), and potential choice (\( P|P \)).

To define the semantics more conveniently, we do not represent actions exactly as they are represented in the graphical diagrams. Instead, we define the set of all actions \( \text{Act} \) by

\[
\text{Act} \overset{\text{def}}{=} \text{IE} \cup \{\epsilon\} \times \text{BExp} \cup \{\epsilon\} \times \text{OE} \cup \{\epsilon\} \times \text{Assign} \cup \{\epsilon\}
\] (7)

where \( \text{IE} \) and \( \text{OE} \) denote the set of all input events and output events, respectively. We require that every action must contain at most one event, i.e.,

\[
(i\epsilon, bx\epsilon, oe\epsilon, a\epsilon) \in \text{Act} \implies (i\epsilon = \epsilon \lor oe\epsilon = \epsilon)
\] (8)

Here we have used \( i\epsilon \) to denote an \( \epsilon \) or an input event, i.e., \( i\epsilon \in \text{IE} \cup \{\epsilon\} \). The same convention is used for boolean expressions, output events, and assignments.

The sets of all input events and output events are defined by

\[
\text{IE} \overset{\text{def}}{=} \{?\} \times \text{M} \quad \text{OE} \overset{\text{def}}{=} \{!\} \times \text{M}
\]

where \( \text{M} \) denotes the set of all messages. An event of the form (\( !, m \)) is an output event representing a transmission of message \( m \), whereas an event of the form (\( ?, m \)) is an input event representing a reception of \( m \). We let \( \text{E} \) denote the set of all events, i.e.,

\[
\text{E} \overset{\text{def}}{=} \text{IE} \cup \text{OE}
\]

Messages are of the form \((nm_t, nm_r, si)\), where \( nm_t \) is the name of the transmitter state machine of the message, \( nm_r \) is the name of the receiver state machine of the message, and \( si \) is the signal of the message. The set of all messages is thus defined by

\[
\text{M} \overset{\text{def}}{=} \text{Nm} \times \text{Nm} \times \text{SI}
\]
We require that the transmitters of output events and receivers of input events in a basic state machine must have the same name. To make this precise, we define the name of an event by the function $nm : E \rightarrow Nm$ as follows:

$$nm(!, (nm_t, nm_r, si)) \overset{def}{=} nm_t$$

$$nm(?,(nm_t, nm_r, si)) \overset{def}{=} nm_r$$

We let $E_{nm}$ denote the set of all events with name $nm$, i.e.,

$$E_{nm} \overset{def}{=} \{ e \in E \mid nm(e) = nm \}$$

We let $nm \in P \rightarrow Nm \cup \{\bot\}$ be the function that yields the name of a state machine expression. The function is defined such that $nm(P) = nm$ if all events in the actions of $P$ are members of $E_{nm}$ for some name $nm$. Otherwise $nm(P) = \bot$.

A basic state machine expression (when we ignore the potential choice) is essentially a regular expression. It is well known that the language of regular expressions is equal to the language defined by a finite state machine [19]. Several procedures exist for deriving a regular expression from a finite state machine (see, e.g., [7, 19, 21]). Although state machine expressions have two kinds of choices (instead of just one as in regular expressions), similar techniques could be used to convert a graphical state machine into a textual state machine expression. Of course, we have to translate action expressions of graphical diagrams to actions of textual state machine expressions, but this is trivial.

As an example, the state machine of Fig. 10 is represented by the following state machine expression:

$$(\epsilon, \epsilon, \epsilon, num = 0) . ((User, Button, pushButton), \epsilon, \epsilon, num = num + 1) . ((\epsilon, \epsilon, (Button, User, display(num))), \epsilon) | (\epsilon, \epsilon, (Button, User, displayHex(num)))^*$$

### A.1.2 Composite state machines

A composite state machine is a state machine that consists of one or more (basic) state machines. There are two constructs for specifying composite state machines: connector and basic state machine reference. The connector is used to indicate that the basic state machines it is connected to may communicate with each other. Connectors are graphically drawn as lines. The basic state machine reference construct is a reference to the basic state machine whose name equals the name of the reference.

An example of a composite state machine is shown in Fig. 11. The composite state machine consists of two basic state machines (User and Button) and one connector indicating that User and Button may communicate with each other.
The notation we use for specifying composite state machines is a subset of the UML composite structure notation.

We are now ready to define the textual syntax of composite state machine expressions.

**Definition A.2 (Composite state machine expression)** The set of all syntactically correct composite state machine expressions $\mathcal{P}^C$ is defined by the following syntax

$$P ::= P_1 \parallel \cdots \parallel P_n$$

where $P_1, \ldots, P_n$ are basic state machine expressions with single distinct names i.e., $\text{nm}(P_i) \neq \bot$ for all $i \in \{1, \ldots, n\}$, and $\text{nm}(P_i) \neq \text{nm}(P_j)$ if $i \neq j$ for all $i, j \in \{1, \ldots, n\}$.

### A.2 Semantics of state machines without potential choice

In this section, we define the semantics of state machines that do not contain the potential choice construct.

#### A.2.1 Basic state machines

In this section, we define the semantics of basic state machine expressions. Roughly speaking, the semantics of a state machine is the set of sequences obtained by recording all input and output events produced in each possible execution of the state machine. To obtain only those sequences that can be produced during execution, we need a way of evaluating expressions, and a way of storing the variables that are assigned to values during execution. The following auxiliary functions are needed to define this more precisely. The definitions of these functions are inspired by [23].

We let $\var \in \text{Exp} \rightarrow \text{P(Var)}$ be the function that yields the set of all variables in an expression. An expression $ex \in \text{Exp}$ is closed if $\var(ex) = \emptyset$.

We let $\text{CExp}$ denote the set of closed expressions, defined as:

$$\text{CExp} \overset{\text{def}}{=} \{ex \in \text{Exp} \mid \var(ex) = \emptyset\}$$

We assume the existence of a function $\text{eval}(\cdot) : \text{CExp} \rightarrow \text{Val}$ that evaluates any closed expression to a value. In the obvious way, we lift $\text{eval}$ to signals containing closed expressions as arguments and to events that contain such signals. In addition, we lift $\text{eval}$ to actions as follows

$$\text{eval}(\langle ie, bx, oe, e \rangle) \overset{\text{def}}{=} (\text{eval}(ie), \text{eval}(bx), \text{eval}(oe), e)$$

$$\text{eval}(\langle ie, bx, oe, (x, ex) \rangle) \overset{\text{def}}{=} (\text{eval}(ie), \text{eval}(bx), \text{eval}(oe), (x, \text{eval}(ex)))$$

Let $\sigma \in \text{Var} \rightarrow \text{Exp}$ be a (total) mapping from variables to expressions. We denote such a mapping $\sigma = \{x_1 \mapsto ex_1, x_2 \mapsto ex_2, \ldots\}$ for distinct $x_1, x_2, \ldots \in \text{Var}$ and for $ex_1, ex_2, \ldots \in \text{Exp}$. If $ex_1, ex_2, \ldots \in \text{Val}$ we call it a data state. We let $\Sigma$ denote the set of all mappings and $\hat{\Sigma}$ denote the set of all data states.

We let $\sigma[x \mapsto ex]$ be the mapping $\sigma$ except that it maps $x$ to $ex$, i.e.,

$$\{x_1 \mapsto ex_1, x_2 \mapsto ex_2, \ldots\}[x \mapsto ex] \overset{\text{def}}{=} \{x_1 \mapsto ex_1, \ldots, x_i \mapsto ex, \ldots\}$$

where $x = x_i$ for some $i \in \{1, \ldots, n\}$.
We generalize \( \sigma[x \mapsto ex] \) to \( \sigma[\sigma'] \) in the following way:

\[
\sigma[[x_1 \mapsto ex_1, \ldots, x_n \mapsto ex_n]] \overset{\text{def}}{=} \sigma[x_1 \mapsto ex_1] \cdots [x_n \mapsto ex_n]
\]

We let \( \sigma(x) \) denote the expression that \( x \) maps to. The mapping is lifted to expressions such that \( \sigma(ex) \) yields the expression obtained from \( ex \) by simultaneously substituting the variables of \( ex \) with the expressions that these variables map to in \( \sigma \). We lift \( \sigma \) to signals and events in the same way, and to actions as follows

\[
\sigma((ie_c, bx_c, oe_c, e)) \overset{\text{def}}{=} \sigma((ie_c), \sigma(bx_c), \sigma(oe_c), e)
\]

\[
\sigma((ie_c, bx_c, oe_c, (x, ex))) \overset{\text{def}}{=} \sigma((ie_c), \sigma(bx_c), \sigma(oe_c), (x, \sigma(ex)))
\]

To define the semantics of a basic state machine, i.e., the sequences of events that can be produced during execution, it is necessary to record the change in data state. We define an intermediate semantics for this purpose. The intermediate semantics is defined in terms of sequences of so-called action-state triples. Action-state triples are of the form

\[
(\text{act}, \sigma, \sigma')
\]

where \( \sigma \) denotes the so-called pre-state before \( \text{act} \) is executed, and \( \sigma' \) denotes the post-state after \( \text{act} \) has been executed, e.g., if \( \text{act} \) contains an assignment \((x, ex)\), then \( \sigma' \) is the state \( \sigma \) except that \( x \) maps to \( ex \).

To define the intermediate semantics, we make use of the concatenation operator \( \bowtie \). If \( s \) and \( t \) are sequences, then \( s \bowtie t \) yields the sequence obtained by concatenating \( s \) and \( t \). We lift the concatenation operator to sequence sets such that \( \Phi \bowtie \Phi' \) yields the set obtained by concatenating all sequences of \( \Phi \) with all sequences of \( \Phi' \). Also, we denote by \( \Phi^n \), the set of sequences obtained by concatenating \( \Phi \) \( n \) times. In particular, \( \Phi^0 \) is defined by \( \{\langle \rangle\} \). For example, we have that \( \Phi^5 \) yields \( \Phi \) concatenated five times, i.e.,

\[
\Phi^5 = \Phi \bowtie \Phi \bowtie \Phi \bowtie \Phi \bowtie \Phi
\]

We are now ready to define the intermediate semantics of state machines.

**Definition A.3 (Intermediate semantics of basic state machines)** The intermediate semantics of a basic state machine expression \( P \), written \( \{P\} \), is defined

\[
\{\langle ie_c, bx_c, oe_c, e \rangle\} \overset{\text{def}}{=} \{\langle \text{eval}(\sigma((ie_c, bx_c, oe_c, e)), \sigma, \sigma) \rangle | \sigma \in \hat{\Sigma}\}
\]

\[
\{\langle ie_c, bx_c, oe_c, (x, ex) \rangle\} \overset{\text{def}}{=} \{\langle \text{eval}(\sigma((ie_c, bx_c, oe_c, (x, ex))), \sigma, \\
\sigma[x \mapsto \text{eval}(\sigma(ex))]) \rangle | \sigma \in \hat{\Sigma}\}
\]

\[
\{P, Q\} \overset{\text{def}}{=} \{P\} \bowtie \{Q\}
\]

\[
\{P + Q\} \overset{\text{def}}{=} \{P\} \cup \{Q\}
\]

\[
\{P^n\} \overset{\text{def}}{=} \bigcup_{i \in \mathbb{N}} \{P\}^i
\]

The intermediate semantics is defined in a modular way in the sense that the meaning of an expression is not affected by the context it appears in. This is achieved by letting actions be evaluated under all possible data states. A consequence of this is that the intermediate semantics describes action sequences that will never be produced during execution. There are two reasons for this:
(a) The sequences of the intermediate semantics may contain actions whose
guards are evaluated to false. During execution, however, these actions
will never occur.

(b) Because the pre-state of a triple is not required to be equal to the post-
state of the triple preceding it, values of the data states may implicitly
change at any time between assignments, while this would never occur
during execution.

If a sequence $s$ of action-state triples does not exhibit problems (a) and (b),
we say that it is well formed, and write $\mathit{wft}(s)$ to indicate this. Formally, well
formed sequences of action-state triples are inductively defined by the following
rules:

$$
\begin{array}{c}
\mathit{wft}(\langle \rangle) \\
\mathit{wft}(((\text{ie}_e, e, a_e), \sigma, \sigma')) \\
\mathit{wft}((s \cup (\langle \text{act}, \sigma', \sigma'' \rangle))) \\
\mathit{wft}((s \cup (\langle \text{act}, \sigma, \sigma' \rangle) \cup (\langle \text{act}, \sigma', \sigma'' \rangle))) \\
\end{array}
$$

We let $\mathcal{A}$ denote the set of all well formed action-state triple sequences, i.e.,

$$
\mathcal{A} \overset{\text{def}}{=} \{ s \in (\text{Act} \times \Sigma \times \Sigma)^* \mid \mathit{wft}(s) \}
$$

We are only interested in the input-output behavior of state machines. To
make this precise we let $\mathit{io} \in (\text{Act} \times \Sigma \times \Sigma)^* \rightarrow E^*$ be the function that takes a
sequence $s$ of action-state triples and produces the sequence of events obtained
from $s$ by removing all data stats, assignments, and guards of $s$. The $\mathit{io}$ function
is formally defined by the following rules:

$$
\begin{array}{ll}
\mathit{io}(\langle \rangle) & \overset{\text{def}}{=} \langle \rangle \\
\mathit{io}(\langle \text{ie}, k, e, a_e, \sigma, \sigma' \rangle) \cup (\langle t \rangle) & \overset{\text{def}}{=} \langle \text{ie} \rangle \cup \mathit{io}(t) \\
\mathit{io}(\langle \text{oc}, e, a_e, \sigma, \sigma' \rangle) \cup (\langle t \rangle) & \overset{\text{def}}{=} \langle \text{oc} \rangle \cup \mathit{io}(t) \\
\mathit{io}(\langle \text{bx}, k, e, a_e, \sigma, \sigma' \rangle) \cup (\langle t \rangle) & \overset{\text{def}}{=} \mathit{io}(t) \\
\end{array}
$$

We lift $\mathit{io}$ to sets of action-state triple sequences such that $\mathit{io}(\Phi)$ yields the set
of sequences obtained by applying $\mathit{io}$ to each sequence in $\Phi$. For example, we
have that

$$
\mathit{io}(\langle \langle \langle \text{?}, m \rangle, b, k, e, \langle x, e x \rangle, \sigma_1, \sigma_1' \rangle, \langle \langle e, b, k', l, m', e \rangle, \sigma_2, \sigma_2' \rangle \rangle) = \\
\langle \langle \langle \langle \text{?}, m \rangle, \langle l, m' \rangle \rangle \rangle
$$

**Definition A.4 (Semantics of basic state machines)** The semantics of a
basic state machine expression $P$, written $\llbracket P \rrbracket$, is defined

$$
\llbracket P \rrbracket \overset{\text{def}}{=} \mathit{io}(\llbracket P \rrbracket \cap \mathcal{A})
$$

**A.2.2 Composite state machines**

We assume that basic state machines are autonomous, and therefore executed
in parallel. Semantically, parallel composition is defined by interleaving traces.

A trace $t$ is an interleaving of the traces $s_1, \ldots, s_n$, written $\mathit{inl}(s_1, \ldots, s_n, t)$, iff $s_1, \ldots, s_n$ are sub-sequences of $t$, i.e.,

$$
\mathit{inl}(\langle \rangle, \langle \rangle, \langle \rangle) \quad \mathit{inl}(\langle s_1, \ldots, s_i, \ldots, s_n, t \rangle) \quad \mathit{inl}(\langle s_1, \ldots, \langle e \rangle \cup s_i, \ldots, s_n, \langle e \rangle \cup t \rangle)
$$
We are only interested in interleavings that are well formed in the sense that they describe communication in which messages are sent before they are received. We write \( wft(s) \) if trace \( s \) is well formed. The predicate is formally defined by the rules:

\[
\begin{align*}
\text{wft}(\langle \rangle) & \quad \text{wft}(s \langle e \rangle \sim t) \quad \text{if } e \in \text{OE} \\
\text{wft}(s \langle !, m \rangle \sim t \langle ?, m \rangle) &
\end{align*}
\]

We let \( T \) denote the set of all well formed traces, i.e.,

\[
T \overset{\text{def}}{=} \{ s \in E^* \mid \text{wft}(s) \}
\]

We define the parallel composition operator as the set of all well formed interleavings of its arguments, i.e.,

\[
\parallel (s_1, \ldots, s_n) \overset{\text{def}}{=} \{ t \in T \mid \text{inl}((s_1, \ldots, s_n), t) \}
\]

(12)

The operator is lifted to trace sets as follows

\[
\parallel (\Phi_1, \ldots, \Phi_n) \overset{\text{def}}{=} \bigcup_{s_1 \in \Phi_1, \ldots, s_n \in \Phi_n} \parallel (s_1, \ldots, s_n)
\]

(13)

**Definition A.5 (Semantics of composite state machines)** The semantics of a composite state machine expression \( P = (P_1 \parallel \ldots \parallel P_n) \), written \([P]\), is defined by

\[
[P] \overset{\text{def}}{=} \parallel ([P_1], \ldots, [P_n])
\]

**A.3 Semantics of state machines with potential choice**

The semantics of a state machine expression with potential choice is a set of trace sets called obligations. Intuitively, the traces of an obligation are equivalent in the sense that an implementation is only required to produce one of them. Thus, each trace of an obligation represents underspecification, i.e., a potential choice. However, an implementation is required to produce at least one trace in each obligation. Thus obligations represent explicit choices that have to be present in an implementation.

To define the semantics formally, we lift the concatenation operator \( \sim \) to sets of obligations as follows

\[
\Omega \sim \Omega' \overset{\text{def}}{=} \{ \phi \sim \phi' \mid \phi \in \Omega \land \phi' \in \Omega' \}
\]

We denote by \( \Omega^n \), the set of obligations \( \Omega \) concatenated \( n \) times. We define \( \Omega^0 \) by \( \{ \langle \rangle \} \).

We define the *inner union* of two sets of obligations, written \( \Omega \uplus \Omega' \), by

\[
\Omega \uplus \Omega' \overset{\text{def}}{=} \{ \phi \uplus \phi' \mid \phi \in \Omega \land \phi' \in \Omega' \}
\]

As we did in the previous section, we first define the intermediate semantics.

**Definition A.6 (Intermediate semantics of basic state machines)** The intermediate semantics of a basic state machine \( P \) with potential choice, written \([\llbracket P\rrbracket]\), is defined
We are now ready to define the semantics of state machine expressions with potential choices.

**Definition A.7 (Semantics of basic state machines)** The semantics of a basic state machine expression $P$ with potential choice, written $[P]$, is defined by

$$[P] \overset{\text{def}}{=} \{ \langle \sigma \rangle \mid \sigma \in \tilde{\Sigma} \}$$

The semantics of a composite state machine expression is the parallel composition of the (basic) state machine expressions it consists of.

**Definition A.8 (Semantics of composite state machines)** The semantics of the composite state machine expression $P = (P_1 \parallel \cdots \parallel P_n)$ with potential choice, written $[P]$, is defined by

$$[P] = \bigcup_{\phi_1 \in [P_1], \ldots, \phi_n \in [P_n]} \{ \parallel (\phi_1, \ldots, \phi_n) \}$$

### B Event transformations

In this section, we give a formal characterization of event transformations and show that these transformations satisfy some desirable properties. In Sect. B.1, we consider transformation of basic state machines, whereas in Sect. B.2, transformation of composite state machines is considered.

#### B.1 Basic state machines

In this section, we give a formal characterization of the transformation which is induced by an event library. The transformation is defined for state machines without potential choices.

An event library is a set of **event specifications**. An event specification is a pair $(e^d, P^p)$ consisting of an event definition $e^d$ and a state machine pattern $P^p$ defining the behavior of the event. An event definition is of the form $(k, si^d)$ where $k$ is a kind (! or ?), and $si^d$ is a **signal definition** of the form $st(f_{p1}, \ldots, f_{pn})$ where $f_{p1}, \ldots, f_{pn}$ are formal parameters. We let $FP$ denote the set of all formal parameters. The set of all signal definitions $SI^d$ and event definitions $E^d$ are defined by

$$SI^d \overset{\text{def}}{=} \text{Str} \times FP^* \quad E^d \overset{\text{def}}{=} \{!, ?\} \times SI^d$$

A state machine pattern $P^p$ is a state machine expression that may contain formal parameters. To make this more precise, we define the notion of expression pattern, event pattern, and action pattern.
An expression pattern $ex^p$ is an expression that may contain formal parameters. We let $Exp^p$ denote the set of all expression patterns, and $BExp^p$ denote the set of all boolean expression patterns. Note that $Exp \subset Exp^p$ and $BExp \subset BExp^p$.

An event pattern $e^p$ is an event whose transmitter and receiver may contain special formal parameters and whose signals may contain expression patterns. The set of all expression patterns $E^p$ is defined by

$$E^p = \{!, ?\} \times ((Nm \cup \{to\}) \times (Nm \cup \{from\}) \times (FP \cup (Str \times (Exp^p)^*)))$$

where $to$ and $from$ are special formal parameters, i.e., $to, from \in FP$.

We let $IE^p$ and $OE^p$ denote the set of all input event patterns and output event patterns, respectively. These sets are defined by

$$IE^p = \{((k,m^p) \in E^p | k = ?)$$

$$OE^p = \{((k,m^p) \in E^p | k = !)$$

The set of all action patterns $Act^p$ is defined by

$$Act^p = (IE^p \cup \{\epsilon\}) \times (BExp^p \cup \{\epsilon\}) \times (OE^p \cup \{\epsilon\}) \times ((Var \times Exp^p) \cup \{\epsilon\})$$

We are now ready to define state machine patterns precisely.

**Definition B.1 (State machine pattern)** The set of all state machine patterns $P^p$ is defined by the following grammar


where $act^p \in Act^p$.

The notion of an event library is made precise by the following definition.

**Definition B.2 (Event library)** The set of all event libraries $EL$ is given by

$$EL = E^d \times P^p$$

For every event library $EL$ in $EL$, we require the signal definitions in $EL$ to be unique in the following sense

$$((k, st(f_{p_1}, \ldots, f_{p_j})), P^p_1), ((k, st(f'_{p_1}, \ldots, f'_{p_k})), P^p_2) \in EL \implies j \neq k$$

In addition, we require that each event specification $((k, (si^d, P^p)))$ in $EL$ must satisfy

$$ffp(P^p) \subseteq ffp(si^d)$$

where the function $ffp$ yields the set of all formal parameters in a signal definition or a state machine pattern.

When a transformation induced by an event library $EL$ is applied to a state machine $P$, all events in $P$ for which there is a matching event definition in $EL$, are replaced by their specification in $EL$. An event $(k, (nm_i, nm_r, si))$ matches an event definition $(k, si^d)$ if there is a substitution $sub$ that replaces formal parameters by expressions such that $si = sub(si^d)$. More precisely, a substitution $sub \in FP \rightarrow Exp$ is a function that replaces formal parameters by expressions. We let $Sub$ denote the set of all substitutions. Any substitution
sub is lifted to expression patterns such that \(\text{sub}(ex^p)\) yields the expression obtained from \(ex^p\) by replacing each formal parameter \(fp\) in \(ex^p\) by \(\text{sub}(fp)\). If \(ex^p\) contains no formal parameters, then \(\text{sub}(ex^p) = ex^p\). Substitutions are further lifted to event patterns and assignment patterns as follows

\[
\text{sub}((k,(nm^p_1, nm^p_2, si(ex^p_1, \ldots, ex^p_n)))) \\
\quad \overset{\text{def}}{=} (k,(\text{sub}(nm^p_1), \text{sub}(nm^p_2), \ldots, \text{sub}(ex^p_n)))
\]

\[
\text{sub}((k,(nm^p_1, nm^p_2, fp))) \\
\quad \overset{\text{def}}{=} (k,(\text{sub}(nm^p_1), \text{sub}(nm^p_2), \text{sub}(fp)))
\]

\[
\text{sub}((x, ex^p)) \\
\quad \overset{\text{def}}{=} (x, \text{sub}(ex^p))
\]

and to state machine patterns as follows

\[
\text{sub}(k,(ie^p_1, bx^p_1, oe^p_1, a^p_1)) \\
\quad \overset{\text{def}}{=} (k,(\text{sub}(ie^p_1), \text{sub}(bx^p_1), \text{sub}(oe^p_1), \text{sub}(a^p_1)))
\]

\[
\text{sub}(P^p_1, P^p_2) \\
\quad \overset{\text{def}}{=} \text{sub}(P^p_1) + \text{sub}(P^p_2)
\]

\[
\text{sub}((P^p)^*) \\
\quad \overset{\text{def}}{=} \text{sub}(P^p)^*
\]

Note that \(ie^p\) denotes an input event pattern or an \(\epsilon\), i.e., \(bx^p \in IE^p \cup \{\epsilon\}\). The same convention is also used for boolean expression patterns, output event patterns, and assignment patterns.

We let \(te_{\text{EL}}(\_\_ ) \in \text{EL} \rightarrow (\text{E} \rightarrow \text{P})\) be the function that replaces events by their definition in an event library. Formally, the function is defined by

\[
te_{\text{EL}}((k,(nm_t,nm_r,si))) \overset{\text{def}}{=} \text{sub}[[\{to \rightarrow nm_t, from \rightarrow nm_r\}](P^p) \\
\quad \text{if there are } ((k,si^i),P^p) \in \text{EL}  \\
\quad \text{and } sub \in Sub \text{ such that } sub(si^i) = si
\]

Note that the special formal parameters \(to\) and \(from\) are bound to the transmitter and the receiver of messages, respectively. As we did in App. A.2, we let \(\text{sub}(fp \mapsto cx)\) denote the mapping \(\text{sub}\) except that \(fp\) maps to \(cx\).

The function \(tact_{\text{EL}}(\_\_ ) \in \text{EL} \rightarrow (\text{Act} \rightarrow \text{P})\) that substitutes events in actions by their specification in an event library is defined as follows

\[
tact_{\text{EL}}((ie,bx_c,\epsilon,a_c)) \overset{\text{def}}{=} (\epsilon,bx_c,\epsilon,\epsilon).te_{\text{EL}}(ie).(\epsilon,\epsilon,\epsilon,a_c) \\
\quad \text{if } ie \in \text{Dom}(te_{\text{EL}})
\]

\[
tact_{\text{EL}}((\epsilon,bx_c,oe,a_c)) \overset{\text{def}}{=} (\epsilon,bx_c,\epsilon,\epsilon).te_{\text{EL}}(oe).(\epsilon,\epsilon,\epsilon,a_c) \\
\quad \text{if } oe \in \text{Dom}(te_{\text{EL}})
\]

\[
tact_{\text{EL}}((ie_c,bx_c,oe_c,a_c)) \overset{\text{def}}{=} (ie_c,bx_c,oe_c,a_c) \\
\quad \text{if } ie_c \notin \text{Dom}(te_{\text{EL}}) \text{ and } oe_c \notin \text{Dom}(te_{\text{EL}})
\]

We let \(T_{\text{EL}}\) be the function that takes a basic state machine \(P\) and yields the state machine \(T_{\text{EL}}(P)\) obtained from \(P\) by replacing the events occurring in each action of \(P\) by their definition in \(EL\). We let \(T_{\text{EL}}\) rename variables of the event specification such that no name clashes occur when events are being replaced by their definitions. To make this precise, we make use of a renaming function \(rnm(\_\_ ) \in \text{P} \times \text{EL} \rightarrow \text{EL}\). That is, \(rnm(P,EL)\) yields the event library obtained from \(EL\) by renaming all its variables such that the following condition is satisfied

\[
\var(rnm(P,EL)) \cap \var(P) = \emptyset
\]
where the function \( \text{var} \) is lifted to event libraries and state machines such that \( \text{var}(EL') \) yields the set of all variables occurring in the state machine patterns of \( EL' \), and \( \text{sub}(P) \) yields the set of all variables occurring in \( P \).

The event transformation \( T_{EL} \) that applies \( \text{tact}_{EL} \) to all actions of a basic state machine is given by the following definition.

**Definition B.3 (Basic event transformation)** The event transformation \( T_{EL}(\cdot) \in EL \rightarrow (P \rightarrow P) \) for basic state machines is defined by

\[
T_{EL}(P) \overset{\text{def}}{=} T_{\text{tnm}(P,EL)}(P)
\]

where \( T_{\cdot}(\cdot) \in EL \rightarrow (P \rightarrow P) \) is defined by

\[
\begin{align*}
T'_{EL}(\text{act}) & \overset{\text{def}}{=} \text{tact}_{EL}(\text{act}) \\
T'_{EL}(P, Q) & \overset{\text{def}}{=} T'_{EL}(P, T'_{EL}(Q)) \\
T'_{EL}(P + Q) & \overset{\text{def}}{=} T'_{EL}(P) + T'_{EL}(Q) \\
T'_{EL}(P^*) & \overset{\text{def}}{=} T'_{EL}(P)^*
\end{align*}
\]

**Definition B.4 (Semantics of basic event transformations)** The semantics of a transformation \( T_{EL} \), written \( \llbracket T_{EL} \rrbracket \), is defined by

\[
\{ \llbracket P \rrbracket, \llbracket T_{EL}(P) \rrbracket \mid P \in \text{Dom}(T_{EL}) \}
\]

Here \( \text{Dom}(T_{EL}) \) yields the domain of the function \( T_{EL} \).

The renaming of variables in event libraries ensures that event specifications are side effect free. This has the consequence that event transformations preserve semantic equality and are therefore functional.

**Theorem B.1** The relation \( \llbracket T_{EL} \rrbracket \) is a function.

The following lemmas tells us that the semantics of an event transformation is entirely characterized by its behavior on events. This property is useful when defining conditions under which event transformations are security preserving.

**Lemma B.1** If \( EL \) is an event library, and \( P \) and \( Q \) be basic state machines whose variables are disjoint from those in \( EL \), then \( \llbracket T_{EL}' \rrbracket \) is homomorphic w.r.t. union, i.e.,

\[
\llbracket T'_{EL} \rrbracket(\llbracket P \rrbracket \cup \llbracket Q \rrbracket) = \llbracket T'_{EL} \rrbracket(\llbracket P \rrbracket) \cup \llbracket T'_{EL} \rrbracket(\llbracket Q \rrbracket)
\]

**Lemma B.2** If \( EL \) is an event library, and \( P \) and \( Q \) be basic state machines whose variables are disjoint from those in \( EL \), then \( \llbracket T'_{EL} \rrbracket \) is homomorphic w.r.t. concatenation, i.e.,

\[
\llbracket T'_{EL} \rrbracket(\llbracket P \rrbracket) \cdot \llbracket T'_{EL} \rrbracket(\llbracket Q \rrbracket) = \llbracket T'_{EL} \rrbracket(\llbracket P \rrbracket) \cdot \llbracket T'_{EL} \rrbracket(\llbracket Q \rrbracket)
\]

**B.2 Composite state machines**

The transformation induced by an event library \( EL \) which takes a composite state machine \( P \) as input, written \( T'_{EL}^{P} \), is simply the function that applies \( T_{EL} \) to all basic state machines that \( P \) consists of.
Definition B.5 (Composite event transformations) The composite transformation $T^C(\_): \in EL \rightarrow (PC \rightarrow PC)$ induced by an event library is defined

$$T_{EL}^C(P_1 \parallel \cdots \parallel P_n \parallel EL) = T_{EL'}^C(P_1) \parallel \cdots \parallel T_{EL'}^C(P_n)$$

where $EL' = \text{rnm}(P_1 \parallel \cdots \parallel P_n, EL)$.

The semantics of $T^C_{EL}$ is defined exactly in the same way as for $T_{EL}$.

Definition B.6 (Semantics of composite event transformations) The semantics of the transformation $T^C_{EL}$, written $\llbracket T^C_{EL} \rrbracket$, is the relation defined by

$$\{(\llbracket P \rrbracket, \llbracket T^C_{EL}(P) \rrbracket) \mid P \in \text{Dom}(T^C_{EL})\}$$

A transformation $T^C_{EL}$ may in general transform non well formed traces resulting from parallel composition at the abstract level into well formed traces at the concrete level. We say that the image of a transformation $T^C_{EL}$, written $\text{Im}_{EL}$, is the set of all concrete traces that have an abstract equivalent, i.e., the set of all concrete traces that correspond to well formed abstract traces. Formally, $\text{Im}_{EL}$ is the least set satisfying the following condition

$$\bigwedge_{i \in \{1, \ldots, n\}}(s_i \in (\text{Enm})^* \land s'_i \in \llbracket T_{EL}(\{s_i\})\rrbracket \land \| (s_1, \ldots, s_n) \neq \emptyset \Rightarrow \| (s'_1, \ldots, s'_n) \subseteq \text{Im}_{EL})$$

for all $\{nm_1, \ldots, nm_n\} \subseteq \text{Nm}$.

The semantics of transformation $T^C_{EL}$ restricted to its image, written $\hat{T}_{EL}$, is defined

$$\hat{T}_{EL} = \{\llbracket P \rrbracket, \llbracket T^C_{EL}(P) \rrbracket \cap \text{Im}_{EL}) \mid P \in \text{Dom}(T^C_{EL})\}$$

Theorem B.2 The relation $\llbracket T^C_{EL} \rrbracket$ is a function when restricted to its image, i.e.,

$$\llbracket P \rrbracket = \llbracket Q \rrbracket \Rightarrow \llbracket T^C_{EL}(P) \rrbracket \cap \text{Im}_{EL} = \llbracket T^C_{EL}(Q) \rrbracket \cap \text{Im}_{EL}$$

Lemma B.3 The semantics of the event transformation for composite state machines induced by an event library $EL$ is homomorphic w.r.t. the union operator on trace sets when restricted to its image, i.e.,

$$\hat{T}_{EL}(\llbracket P \rrbracket \cup \llbracket Q \rrbracket) = \hat{T}_{EL}(\llbracket P \rrbracket) \cup \hat{T}_{EL}(\llbracket Q \rrbracket)$$

B.3 Adherence preservation under event transformations

In this section, we present general conditions under which security properties are preserved under event transformations. We then show how these general conditions can be instantiated into specific conditions for particular security properties.

We recall from [34, 33], that a security property $\text{Sec}P$ is a conjunction of basic security predicates $\text{BSP}_{\text{rb}}(\Phi)$ of the form

$$\forall s, t \in \Phi : r s \rightarrow t \Rightarrow \exists u \in \Phi : \text{h} s \rightarrow u \sim_{\text{c}} t$$

for the restriction relation $\text{r}$, the high level relation $\text{h}$, and the low-level equivalence relation $\sim_{\text{c}}$ on traces (i.e., $\text{r}, \text{h}, \sim_{\text{c}} \subseteq \text{E}^* \times \text{E}^*$). Note that if $\text{c}$ is a relation, we write $s \sim_{\text{c}} t$ for $(s, t) \in \text{c}$, and $s \sim_{\text{c}} t$ for $(s, t) \in \text{c}$ if $\text{c}$ is an equivalence relation.
We will henceforth use the following definition of the low level equivalence relation
\[ s \sim_L t \iff s|_L = t|_L \]  
for a set of low level events \( L \).

A transformation is secure w.r.t. a security property \( \text{SecP} \) if the image of an abstract specification \( P \) is secure w.r.t. \( \text{SecP} \) if \( P \) is secure w.r.t. \( \text{SecP} \).

We restrict attention to the image since we cannot exploit the fact that the abstract specification is secure to ensure that concrete traces that do not have any abstract equivalent do not violate security. This means that additional security analysis is needed at the concrete level to ensure that those traces that do not have any abstract equivalent do not violate security.

**Definition B.7 (Security preservation)** Let \( EL \) be an event library and \( \text{SecP} \) be a security property. Then transformation \( T^C_{EL} \) induced by \( EL \) preserves security property \( \text{SecP} \) for specification \( P \) iff
\[ \text{SecP}(\llbracket P \rrbracket) \implies \text{SecP}(\hat{T}_{EL}(\llbracket P \rrbracket)) \]

The following theorem presents general conditions under which a basic security predicate is preserved by a transformation.

**Theorem B.3** Let \( r \) be a restriction, \( h \) be a high level relation, and \( T^C_{EL} \) be the transformation induced by event library \( EL \). Then \( T^C_{EL} \) preserves \( \text{Bsp}_{rh} \) for specification \( P \) if the following conditions are satisfied for all \( s, t, u \in \llbracket P \rrbracket, s' \in \hat{T}_{EL}(\{s\}), t' \in \hat{T}_{EL}(\{t\}) \):

\[ s' \xrightarrow{r} t' \implies s \xrightarrow{r} t \]  
\[ s \xrightarrow{r} t \wedge s \xrightarrow{h} u \sim_L t \implies \exists u' \in \hat{T}_{EL}(\{u\}) : s' \xrightarrow{h} u' \sim_L t' \]  

Note that we have exploited Lemma B.3 in the definition of the conditions since the property ensures that the transformation \( \hat{T}_{EL} \) is defined for singleton sets. Obviously, a transformation preserves a security property \( \text{SecP} \) if the conditions are satisfied for all basic security predicates that \( \text{SecP} \) consists of.

**Corollary B.1** Let \( \text{SecP} \) be a security property, i.e., a conjunction of basic security predicates. Then \( \text{SecP} \) is preserved under a transformation \( T^C_{EL} \) if each basic security predicate of \( \text{SecP} \) satisfies the conditions (20) and (21) of Theorem B.3.

The conditions of Theorem B.3 may be used to derive specific conditions that can be used to check that a transformation preserves a given security property. The procedure for deriving such conditions for an arbitrary security property \( \text{SecP} \) and event library \( EL \) is as follows. For each basic security predicate \( \text{Bsp}_{rh} \) that \( \text{SecP} \) consists of:

- define two conditions \( C_1 \) and \( C'_1 \) and show that \( C_1 \) implies (20) and \( C'_1 \) implies (21) for the transformation \( T_{EL} \) of basic state machines induced by \( EL \);
• define two conditions \( C_2 \) and \( C'_2 \) and show that \( C_1 \land C_2 \) implies (20) and \( C'_2 \land C'_2 \) implies (21) for the composite transformation \( T_{EL}^C \) induced by \( EL \).

In the following we illustrate the procedure for the non-inference property [30].

**Definition B.8** The security property non-inference, written \( NF \), consists of the basic security predicate \( BSP_{ch} \) whose restriction \( r \) and high level relation \( h \) are defined

\[
\begin{align*}
    s \xrightarrow{r} t & \iff \text{true} \\
    s \xrightarrow{h} t & \iff t|_H = \langle \rangle
\end{align*}
\]

According to the procedure, we must first define two conditions \( C_1 \) and \( C'_1 \) and show that \( C_1 \) implies (20) and \( C'_1 \) implies (21), respectively. By definition of \( r \), any transformation will trivially satisfy condition (20). Thus we need only consider the latter condition (21). We first instantiate (21) for the basic security predicate defining the \( NF \) property. We get for all traces \( t \) and \( u \), and all \( t' \in \widehat{T_{EL}}(\{t\}) \)

\[
   u|_H = \langle \rangle \land u|_L = t|_L \implies \exists u' \in \widehat{T_{EL}}(\{u\}) : u'|_H = \langle \rangle \land u'|_L = t'|_L \tag{22}
\]

To define a condition \( C'_1 \) that implies (22) for basic transformations, we first examine the case in which \( \llbracket T_{EL} \rrbracket \) is applied to single events. We can do so because \( \llbracket T_{EL} \rrbracket \) is homomorphic w.r.t. union and concatenation of trace sets (by Lemma B.1 and Lemma B.2). It is easy to see that any transformation that transforms a non high level event into a set of traces that contain a high level event will fail to satisfy condition (22). Hence, we define a condition that ensures that this never occurs:

\[
    e \notin H \implies \forall s \in \llbracket T_{EL} \rrbracket(\{e\}) : s|_H = \langle \rangle \tag{23}
\]

The same consideration holds for low level events; if a non low level event is transformed into a set of traces that contain a low level event, then condition (22) is broken. Therefore we must require

\[
    e \notin L \implies \forall s \in \llbracket T_{EL} \rrbracket(\{e\}) : s|_L = \langle \rangle \tag{24}
\]

When the set \( L \) of low level events, and the set \( H \) of high level events are defined as all events that may occur in particular state machines, then conditions (23) and (24) are trivially satisfied because event transformations cannot change the transmitters or receivers of events. By exploiting Lemma B.2, it follows by induction over the length of traces \( u \) and \( t \) that any transformation on basic state machines satisfying conditions (23) and (24) will satisfy the condition (22).

The final step of the procedure is to define a condition \( C'_2 \) and to show that this condition together with (23) and (24) implies (22) for transformations of composite state machines. To obtain \( C'_2 \) we observe that the first part of condition (22) \( (u'|_H = \langle \rangle) \) is ensured by (23). However, this is not the case for the second part \( (u'|_L = t'|_L) \) which requires low level equality to be preserved. This condition can be rephrased as

\[
    \widehat{T_{EL}}(\{t|_L\}) = \widehat{T_{EL}}(\{t\})|_L \tag{25}
\]

for all traces \( t \).
The reason why condition (25) may not be satisfied is that low level observations on the abstract level may be transformed into several concrete low level observations that may be part of traces that are not well formed. To see this, consider the upper most composite state machine of Fig. 12. Let the set of low level events \( L \) be the set of events that occur in state machine \( P_1 \), and the set \( H \) of high level events be \( \{?d, ?f\} \). The semantics of \( \mathcal{A} \) is

\[
[\mathcal{A}] = \langle \{!a\}, \{?a, !c, ?a, !d\} \rangle = \{\{!a, ?a, !c, ?c\}, \{!a, ?a, !d, ?d\}\}
\]

The specification is secure w.r.t. the \( \text{Nf} \)-property because when the low level user observes \( !a \) (this is the only observation the low level user can make), he will not know whether the high level event \(?d\) will occur.

Now let \( EL \) be an event library defined such that \( T_{EL}^C(\mathcal{A}) = \mathcal{C} \) where \( \mathcal{C} \) is the lower most specification of Fig. 12. Specification \( \mathcal{C} \) has the following semantics

\[
[\mathcal{C}] = \langle \{!a_1, !a_1\}, \{!a_2, !a_2\} \rangle = \{\{!a_1, ?a_1, ?a_1, !f\}, \{!a_2, ?a_1, !f\}, \{?a_2, ?a_2, !f\}\}
\]

The specification \( \mathcal{C} \) is not secure w.r.t. the \( \text{Nf} \) property because the low level user will know that the high level event \(?f\) occurs when the observation \( !a_2, !a_2 \) is made. The problem is that the low level observation \( !a \) has been transformed into the two observations \( !a_1, !a_1 \) and \( !a_2, !a_2 \). But the trace in which both observation \( !a_2, !a_2 \) and event \( !e \) (instead of the high level event \( !f \)) occurs is not well formed, and therefore not part of the concrete specification.

The check that condition (25) is satisfied for a state machine \( P \), we may for instance, check that the traces obtained by transforming two corresponding events of the event library are not affected by the context they appear in, i.e.,

\[
((!, m), (?, m)) \in \text{Dom}(te_{EL}) \land t \in [P] \land (\langle !, m \rangle, \langle ?, m \rangle) \prec t \implies ((s' \in T_{EL}([\langle !, m \rangle, \langle ?, m \rangle])) \iff (\exists t' \in T_{EL}([t]) : s' \prec t'))
\]

where \( s \prec t \) means that \( s \) is a sub-trace of \( t \), i.e., a trace \( s = (e_1, \ldots, e_n) \) is a sub-trace of \( t \) iff

\[
s_1 \prec e_1 \prec \cdots \prec s_n \prec (e_n) \prec s_{n+1} = t
\]

for some \( s_1, \ldots, s_{n+1} \in E^* \).

The conditions (23), (24), and (25) implies that (22) holds. By Theorem B.3 and definition of \( \text{Nf} \), this means that any transformation satisfying (23), (24), and (25) will preserve the \( \text{Nf} \) property.
C Syntactic categories

<table>
<thead>
<tr>
<th>Set</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{P}(A)$</td>
<td>${X</td>
</tr>
<tr>
<td>$A^*$</td>
<td>The set of all finite sequences over the set $A$.</td>
</tr>
<tr>
<td>$ax \in AExp$</td>
<td>Arithmetic expressions</td>
</tr>
<tr>
<td>$bx \in BExp$</td>
<td>Boolean expressions</td>
</tr>
<tr>
<td>$sx \in SExp$</td>
<td>String expressions</td>
</tr>
<tr>
<td>$ex \in Exp$</td>
<td>$\text{Exp} \cup \text{AExp} \cup \text{BExp} \cup \text{SExp}$ Expressions</td>
</tr>
<tr>
<td>$Val \subseteq Exp$</td>
<td>The set of all values.</td>
</tr>
<tr>
<td>$x, y \in \text{Var} \subseteq Exp$</td>
<td>The set of all variables.</td>
</tr>
<tr>
<td>$a \in \text{Assign} \subseteq \text{Var} \times \text{Exp}$</td>
<td>The set of all assignments.</td>
</tr>
<tr>
<td>$nm \in \text{Nm}$</td>
<td>The set of all names.</td>
</tr>
<tr>
<td>$si \in \text{SI}$</td>
<td>The set of all signals.</td>
</tr>
<tr>
<td>$m \in M \subseteq \text{Nm} \times \text{Nm} \times \text{SI}$</td>
<td>The set of all messages.</td>
</tr>
<tr>
<td>$ie \in \text{IE} \subseteq {?} \times M$</td>
<td>The set of all input events.</td>
</tr>
<tr>
<td>$oe \in \text{OE} \subseteq {!} \times M$</td>
<td>The set of all output events.</td>
</tr>
<tr>
<td>$e \in E \subseteq \text{IE} \cup \text{OE}$</td>
<td>The set of all events.</td>
</tr>
<tr>
<td>$act \in \text{Act} \subseteq \text{IE} \cup {\epsilon} \times \text{BExp} \cup {\epsilon} \times \text{OE} \cup {\epsilon} \times \text{Assign} \cup {\epsilon}$</td>
<td>The set of all actions.</td>
</tr>
</tbody>
</table>

$P \in \mathcal{P}$ Set of all basic state machine expressions.

$P \in \mathcal{P}^C$ Set of all composite state machine expressions.

* By the notation $a \in A$ we understand that $A$ is ranged over by $a$. 

Figure 12: Example
D Proofs

D.1 Auxiliary definitions

In this section, we make some definitions that are needed in the proofs.

Definition D.1 We let the function $|\_| : P \rightarrow P(\text{Act}^*)$ yield the set of action sequences described by a basic state machine without potential choice. This function is defined by

- $|\text{act}| = \{\langle \text{act} \rangle\}$
- $|P.Q| \overset{\text{def}}{=} |P| \leftarrow |Q|$
- $|P + Q| \overset{\text{def}}{=} |P| \cup |Q|$
- $|P^*| \overset{\text{def}}{=} \bigcup_{i \in \mathbb{N}} (|P|)^i$

Definition D.2 We let the functions $\text{preState}(\_), \text{postState}(\_) : (\text{Act} \times \hat{\Sigma} \times \hat{\Sigma})^+ \rightarrow \hat{\Sigma}$ yield the first and the last state in a non-empty sequence of action-state triples, respectively. These functions are defined by

- $\text{preState}(\langle (\text{act}, \sigma, \sigma') \rangle) \overset{\text{def}}{=} \sigma$
- $\text{postState}(\langle (\text{act}, \sigma, \sigma') \rangle) \overset{\text{def}}{=} \sigma'$

In addition, we let the functions $\text{preState}(\_, V), \text{postState}(\_, V) : (\text{Act} \times \hat{\Sigma} \times \hat{\Sigma})^+ \times \mathbb{P}(\text{Var}) \rightarrow \hat{\Sigma}$ be defined by

- $\text{preState}(\langle \text{as}, V \rangle) \overset{\text{def}}{=} \text{preState}(\langle \text{as} \rangle) \setminus (V \times \text{Val})$
- $\text{postState}(\langle \text{as}, V \rangle) \overset{\text{def}}{=} \text{postState}(\langle \text{as} \rangle) \setminus (V \times \text{Val})$

Definition D.3 We let $pr(\_)$ be the function that yields the prefix closure of a set of sequences. The function is defined by

- $pr(A) \overset{\text{def}}{=} \{s \mid t \in A \land s \sqsubseteq t\}$

where $\sqsubseteq$, the prefix predicate, is defined by

- $s \sqsubseteq t \iff \exists u : s \leftarrow u = t$

D.2 Basic transformations

Theorem B.1 The relation $[T_{\text{EL}}]$ is a function.

Proof of Theorem B.1

Assume: 1. $[P] = [Q]$ for some $P, Q \in P$
- 2. $EL \in \text{EL}$

Prove: $[T_{\text{EL}}(P)] = [T_{\text{EL}}(Q)]$

(1) Assume: 1.1 $EL_1 = \text{rnm}(P, EL)$ and $EL_2 = \text{rnm}(Q, EL)$ for $EL_1, EL_2 \in \text{EL}$

Prove: $[T'_{EL_1}(P)] = [T'_{EL_2}(Q)]$

(2) Assume: 2.1 $s' \in [T'_{EL_1}(P)]$

Prove: $s' \in [T'_{EL_2}(Q)]$

(3) Assume: Choose $as'_1 \in \langle T'_{EL_1}(P) \rangle \cap A$ such that $io(as'_1) = s'$
PROOF: By assumption 2.1 and definition of $\llbracket \cdot \rrbracket$ (Def. A.4).

(3)2. Choose $(act_{1,1}, \ldots, act_{1,j}) \in [P]$ and $as'_{2,1} \in \llbracket T'_{EL_1}(act_{1,1}) \rrbracket, \ldots, as'_{1,j} \in \llbracket T'_{EL_1}(act_{1,j}) \rrbracket$ such that $as'_{1} = as'_{1,1} \ldots \ldots \ldots as'_{1,j}$

PROOF: By (3)1 and Lemma T.B.1.1.

(3)3. Choose $as_1 \in A$ and $as_{1,1} \in \llbracket act_{1,1} \rrbracket, \ldots, as_{1,j} \in \llbracket act_{1,j} \rrbracket$ such that $as_1 = as_{1,1} \ldots \ldots \ldots as_{1,j}$ and $\forall i \in \{1, \ldots, j\}$: $\text{preState}(as_{1,i}, \text{var}(EL_1)) = \text{preState}(as'_{1,i}, \text{var}(EL_1)) \land \text{postState}(as_{1,i}, \text{var}(EL_1)) = \text{postState}(as'_{1,i}, \text{var}(EL_1))$

PROOF: By assumptions 1.1 and 2, (3)1, (3)2, and Lemma T.B.1.2.

(3)4. $\text{io}(as_1) \in \llbracket P \rrbracket$

PROOF: By (3)2, (3)3, definition of $\downarrow$ (Def. A.3), definition of $\uparrow$ (Def. A.4), and definition of $\text{var}$ and $\text{rmn}$ (App. B.1).

(3)5. Choose $as_2 \in \llbracket Q \rrbracket \cap A$ such that $\text{io}(as_2) = \text{io}(as_1)$

PROOF: By (3)4, assumption 1 and definition of $\uparrow$ (Def. A.4).

(3)6. Choose $(act_{2,1}, \ldots, act_{2,k}) \in [Q]$ and $as_{2,1} \in \llbracket act_{2,1} \rrbracket, \ldots, as_{2,k} \in \llbracket act_{2,k} \rrbracket$ such that $as_2 = as_{2,1} \ldots \ldots \ldots as_{2,k}$

PROOF: By (3)5, definition of $\downarrow$ (Def. D.1) and definition of $\downarrow$ (Def. A.3).

(3)7. Choose $as'_{2} \in \llbracket T'_{EL_2}(act_{2,1}) \rrbracket \ldots \ldots \llbracket T'_{EL_2}(act_{2,k}) \rrbracket$ such that $as'_{2} \in A$ and $\text{io}(as'_{2}) = \text{io}(as'_{1})$

PROOF: By assumption 1 and 1.1, (3)1, (3)2, (3)3, (3)4, (3)5, (3)6, and Lemma T.B.1.3.

(3)8. $as'_{2} \in \llbracket T'_{EL_2}(Q) \rrbracket \cap A$

PROOF: By (3)6, (3)7, and definition of $\downarrow$ (Def. A.3) and definition of $T'_{EL_2}$ (Def. B.3).

(3)9. Q.E.D.

PROOF: By (3)1, (3)7, (3)8 and definition of (Def. A.4).

(2)2. ASSUME: $1.1 s' \in \llbracket T'_{EL_2}(Q) \rrbracket$

PROOF: $s' \in \llbracket T'_{EL_1}(P) \rrbracket$

PROOF: By (2)1 and symmetry of $=.$

(2)3. Q.E.D.

PROOF: By (2)1 and (2)2.

(1)2. Q.E.D.

PROOF: By (1)1 and definition of $T_{EL}$ (Def.B.3).

Lemma T.B.1.1 If

- $as' \in \llbracket T_{EL}(P) \rrbracket$

then

- $\exists (act_1, \ldots, act_n) \in [P]: \exists as'_{1} \in \llbracket T_{EL}(act_1) \rrbracket: \ldots: \exists as'_{n} \in \llbracket T_{EL}(act_n) \rrbracket: as' = as'_{1} \ldots \ldots \ldots as'_{n}$

Proof of Lemma T.B.1.1

ASSUME: 1. $EL \in EL$ and $P \in P$

PROVE: $as' \in \llbracket T_{EL}(P) \rrbracket \iff (\exists (act_1, \ldots, act_n) \in [P]: \exists as'_{1} \in \llbracket T_{EL}(act_1) \rrbracket: \ldots: \exists as'_{n} \in \llbracket T_{EL}(act_n) \rrbracket: as' = as'_{1} \ldots \ldots \ldots as'_{n})$

(1)1. CASE: 1.1 $P = act$

(2)1. ASSUME: 2.1 $as' \in \llbracket T_{EL}(act) \rrbracket$

PROVE: $\exists (act_1, \ldots, act_n) \in [act]: \exists as'_{1} \in \llbracket T_{EL}(act_1) \rrbracket: \ldots: \exists as'_{n} \in \llbracket T_{EL}(act_n) \rrbracket: as' = as'_{1} \ldots \ldots \ldots as'_{n}$
(3.1) Choose \( \langle \text{act}_1 \rangle \in \text{[act]} \)
Proof: By definition of \( \langle \cdot \rangle \) (Def. D.1).

(3.2) \( \text{as}' \in \{ T'_{EL}(\text{act}_1) \} \)
Proof: By (3.1) and assumption 2.1.

(3.3) Q.E.D.
Proof: By (3.1) and (3.2).

(2.2) Q.E.D.
Proof: By (2.1) and logical implication.

(1.2) Case: \( 1. \ P = P_1, P_2 \)
1.2 \( \text{as}' \in \{ T'_{EL}(P_1) \} \implies (\exists \langle \text{act}_1, \ldots, \text{act}_n \rangle \in |P_1| : \exists \text{as}'_1 \in \{ T'_{EL}(\text{act}_1) \} : \ldots : \exists \text{as}'_n \in \{ T'_{EL}(\text{act}_n) \} : \text{as}' = \text{as}'_1 \land \cdots \land \text{as}'_n \)

1.3 \( \text{as}' \in \{ T'_{EL}(P_2) \} \implies (\exists \langle \text{act}_1, \ldots, \text{act}_n \rangle \in |P_2| : \exists \text{as}'_1 \in \{ T'_{EL}(\text{act}_1) \} : \ldots : \exists \text{as}'_n \in \{ T'_{EL}(\text{act}_n) \} : \text{as}' = \text{as}'_1 \land \cdots \land \text{as}'_n \)

(2.1) Assume: \( 2.1 \ \text{as}' \in \{ T'_{EL}(P_1, P_2) \} \)
Proof: \( \exists \langle \text{act}_1, \ldots, \text{act}_n \rangle \in |P_1, P_2| : \exists \text{as}'_1 \in \{ T'_{EL}(\text{act}_1) \} : \ldots : \exists \text{as}'_n \in \{ T'_{EL}(\text{act}_n) \} : \text{as}' = \text{as}'_1 \land \cdots \land \text{as}'_n \)

(3.1) Choose \( \langle \text{act}_1, \ldots, \text{act}_n \rangle \in |P_1, P_2| \) and \( \langle \text{as}'_1, \ldots, \text{as}'_n \rangle \) such that \( \text{as}' = \text{as}'_1 \land \cdots \land \text{as}'_n \)
Proof: By assumption 2.1, definition of \( T'_{EL} \) (Def. B.3), and definition of \( \langle \cdot \rangle \) (Def. A.3).

(3.2) Choose \( \langle \text{act}_{1,1}, \ldots, \text{act}_{1,j} \rangle \in |P_1| \) and \( \langle \text{as}'_{1,1}, \ldots, \text{as}'_{1,j} \rangle \) such that \( \text{as}'_1 = \text{as}'_{1,1} \land \cdots \land \text{as}'_{1,j} \)
Proof: By (3.1) and assumption 1.2.

(3.3) Choose \( \langle \text{act}_{2,1}, \ldots, \text{act}_{2,k} \rangle \in |P_2| \) and \( \langle \text{as}'_{2,1}, \ldots, \text{as}'_{2,k} \rangle \) such that \( \text{as}'_2 = \text{as}'_{2,1} \land \cdots \land \text{as}'_{2,k} \)
Proof: By (3.1) and assumption 1.3.

(3.4) \( \langle \text{act}_{1,1}, \ldots, \text{act}_{1,j}, \text{act}_{2,1}, \ldots, \text{act}_{2,k} \rangle \in |P_1, P_2| \)
Proof: By (3.2), (3.3) and definition of \( \langle \cdot \rangle \) (Def. D.1).

(3.5) Q.E.D.
Proof: By (3.1) and (3.4).

(2.2) Q.E.D.
Proof: By (2.1) and logical implication.

(1.3) Case: \( 1. \ P = P_1 + P_2 \)
1.2 \( \text{as}' \in \{ T'_{EL}(P_1) \} \implies (\exists \langle \text{act}_1, \ldots, \text{act}_n \rangle \in |P_1| : \exists \text{as}'_1 \in \{ T'_{EL}(\text{act}_1) \} : \ldots : \exists \text{as}'_n \in \{ T'_{EL}(\text{act}_n) \} : \text{as}' = \text{as}'_1 \land \cdots \land \text{as}'_n \)

1.3 \( \text{as}' \in \{ T'_{EL}(P_2) \} \implies (\exists \langle \text{act}_1, \ldots, \text{act}_n \rangle \in |P_2| : \exists \text{as}'_1 \in \{ T'_{EL}(\text{act}_1) \} : \ldots : \exists \text{as}'_n \in \{ T'_{EL}(\text{act}_n) \} : \text{as}' = \text{as}'_1 \land \cdots \land \text{as}'_n \)

(2.1) Assume: \( 2.1 \ \text{as}' \in \{ T'_{EL}(P_1 + P_2) \} \)
Proof: \( \exists \langle \text{act}_1, \ldots, \text{act}_n \rangle \in |P_1 + P_2| : \exists \text{as}'_1 \in \{ T'_{EL}(\text{act}_1) \} : \ldots : \exists \text{as}'_n \in \{ T'_{EL}(\text{act}_n) \} : \text{as}' = \text{as}'_1 \land \cdots \land \text{as}'_n \)

(3.1) Choose \( i \in \{ 1, 2 \} \) such that \( \text{as}'_i \in \{ T'_{EL}(P_i) \} \)
Proof: By assumption 2.1 and definition of \( T'_{EL} \) (Def. B.3), and definition of \( \{ \} \) (Def. A.3).

(3.2) Choose \( \langle \text{act}_1, \ldots, \text{act}_n \rangle \in |P_1| \) and \( \langle \text{as}'_1, \ldots, \text{as}'_n \rangle \) such that \( \text{as}' = \text{as}'_1 \land \cdots \land \text{as}'_n \)
Proof: By (3.1) and assumption 1.2 or assumption 1.3.

(3.3) \( \langle \text{act}_1, \ldots, \text{act}_n \rangle \in |P_1 + P_2| \)
Proof: By (3.2) and \( \langle \cdot \rangle \) (Def. D.1).

(3.4) Q.E.D.
Proof: By (3.1) and (3.3).
Lemma T.B.1.2

If

- \(\text{var}(EL) \cap (\text{var}(act_1) \cup \ldots \cup \text{var}(act_n)) = \emptyset\)
- \(as' \in A\)
\[ D \) PROOFS \]

- \( as'_1 \in \langle T'_{EL}(act_1) \rangle, \ldots, as'_n \in \langle T'_{EL}(act_n) \rangle \)
- \( as' = as'_1 \cdots as'_n \)

then

- \( \exists as \in A : \exists as_1 \in \langle act_1 \rangle : \ldots : \exists as_n \in \langle act_n \rangle : as = as_1 \cdots as_n \land \\
( \forall i \in \{1, \ldots, n\} : \text{preState}(as_i, var(EL)) = \text{preState}(as'_i, var(EL)) \land \\
\text{postState}(as_i, var(EL)) = \text{postState}(as'_i, var(EL))) \)

**Proof of Lemma T.B.1.2** The proof is by induction on the length of the action sequence \((act_1, \ldots, act_n)\). In the proof we make use of the following definition which highlights the induction.

\[
\text{Ind}((act_1, \ldots, act_n), EL) \overset{\triangle}{=} \\
\forall as' \in A : \forall as'_1 \in \langle T'_{EL}(act_1) \rangle : \ldots : \forall as'_n \in \langle T'_{EL}(act_n) \rangle : \\
as' = as'_1 \cdots as'_n \\
\implies \exists as \in A : \\
\exists as_1 \in \langle act_1 \rangle : \ldots \exists as_n \in \langle act_n \rangle : \\
\land as = as_1 \cdots as_n \\
\land \forall i \in \{1, \ldots, n\} : \\
\land \text{preState}(as_i, var(EL)) = \text{preState}(as'_i, var(EL)) \\
\land \text{postState}(as_i, var(EL)) = \text{postState}(as'_i, var(EL))
\]

Also, if \( s \) is a sequence, we denote by \( \text{pref}(s, j) \), the prefix of \( s \) of length \( j \). If \( j \) is greater than the length of \( s \), then \( \text{pref}(s, j) = s \).

The proof is given by the following.

**Assume:** 1. \( \text{var}(EL) \cap (\text{var}(act_1) \cup \cdots \cup \text{var}(act_n)) = \emptyset \) for \( EL \in EL \) and \( act_1, \ldots, act_n \in \text{Act} \) and \( n \geq 1 \)

**Prove:** \( \text{Ind}((act_1, \ldots, act_n), EL) \)

(\( 1 \))1. Case: 1.1 \( n = 1 \)

(\( 2 \))1. Assume: 2.1 \( as' \in A \)

2.2 \( as'_1 \in \langle T'_{EL}(act_1) \rangle \)

2.3 \( as' = as'_1 \)

**Prove:** \( \exists as \in A : \exists as_1 \in \langle act_1 \rangle : as = as_1 \land \text{preState}(as_1, var(EL)) = \text{preState}(as'_1, var(EL)) \land \text{postState}(as_1, var(EL)) = \text{postState}(as'_1, var(EL))) \)

(\( 3 \))1. Case: 3.1 \( act_1 = (\epsilon, b_{x_\epsilon}, \epsilon, a_\epsilon) \) for \( b_{x_\epsilon} \in B\text{Exp} \cup \{\epsilon\} \) and \( a_\epsilon \in \text{Assign} \cup \{\epsilon\} \)

(\( 4 \))1. Choose \( as \in \langle act_1 \rangle \) such that \( as = as' \)

(\( 5 \))1. \( \langle T'_{EL}(act_1) \rangle = \langle tact_{EL}(act_1) \rangle = \langle act_1 \rangle \)

**Proof:** By assumption 3.1, definition of \( T'_{EL} \) (Def. B.3), and definition of \( tact_{EL} \) (Eq. 15), definition of \( te_{EL} \) (Eq.14), and definition of \( E \) (App. A.1.1).

(\( 5 \))2. Q.E.D.

**Proof:** By assumption 2.2, assumption 2.3, and (\( 5 \))1.

(\( 4 \))2. \( as \in A \)

**Proof:** By (\( 4 \))1 and assumption 2.1.

(\( 4 \))3. \( \text{preState}(as, var(EL)) = \text{preState}(as', var(EL)) \) and \( \text{postState}(as, var(EL)) = \text{postState}(as', var(EL)) \)

**Proof:** By (\( 4 \))1.

(\( 4 \))4. Q.E.D.
Proof: By (4)1, (4)2, and (4)3.

(3) Case: 3.1 \( \text{act}_1 = (ie, bx_e, e, a_e) \) for \( ie \in \textbf{IE} \), \( bx_e \in \textbf{BExp} \cup \{e\} \), and 
\( a_e \in \textbf{Assign} \cup \{e\} \)

(4)1. Case: 4.1 \( ie \in \text{Dom}(te_EL) \)
(5)1. Choose \( \text{ass} \in \{\text{act}_1\} \) such that \( \text{ass} = \text{ass}' \)
(6)1. \( \{T'_EL(\text{act}_1)\} = \{tact_EL(\text{act}_1)\} = \{\text{act}_1\} \)
Proof: By assumption 2.2, assumption 3.1, assumption 4.1, and 
definition of \( T'_EL \) (Def. B.3), and \( \{\_\} \) (Def. A.3).

(5)2. Choose \( \text{as} \in \{\{ie, bx_e, e, a_e\}\} \) such that \( \text{preState}(\text{as}, \text{var}(EL)) = \text{preState}(\text{as}', \text{var}(EL)) \) and 
\( \text{postState}(\text{as}, \text{var}(EL)) = \text{postState}(\text{as}', \text{var}(EL)) \)
Proof: By assumption 1 and (5)1, definition of \( \{\_\} \) (Def. A.3), 
definition of \( te_EL \) (Eq. 14), and definition of \( \text{var}(EL) \) (Section B.1).

(5)3. \( \text{as} \in \mathcal{A} \)
Proof: By assumptions 2.1, 2.2, and 2.3, and (5)1, (5)2, and 
definition of \( \mathcal{A} \) (Eq. 10).

(5)4. Q.E.D.
Proof: By (5)2, assumption 3.1, and (5)3.

(4)2. Case: 4.1 \( ie \notin \text{Dom}(te_EL) \)
(5)1. Choose \( \text{ass} \in \{\text{act}_1\} \) such that \( \text{ass} = \text{ass}' \)
(6)1. \( \{T'_EL(\text{act}_1)\} = \{tact_EL(\text{act}_1)\} = \{\text{act}_1\} \)
Proof: By assumption 3.1, assumption 4.1, definition of \( T'_EL \) 
(Def. B.3), definition of \( tact_EL \) (Eq. (15)), definition of \( te_EL \) (Eq. 14), and 
definition of \( E \) (App. A.1.1).

(6)2. Q.E.D.
Proof: By assumption 2.2, assumption 2.3, and (6)1.

(5)2. \( \text{as} \in \mathcal{A} \)
Proof: By (5)1 and assumption 2.1.

(5)3. \( \text{preState}(\text{as}, \text{var}(EL)) = \text{preState}(\text{as}', \text{var}(EL)) \) and 
\( \text{postState}(\text{as}, \text{var}(EL)) = \text{postState}(\text{as}', \text{var}(EL)) \)
Proof: By (5)2.

(5)4. Q.E.D.
Proof: By (5)1, (5)2, and (5)3.

(4)3. Q.E.D.
Proof: By (4)1 and (4)2.

(3)3. Case: 3.1 \( \text{act}_1 = (ie, bx_e, oe, a_e) \) for \( oe \in \textbf{OE} \), \( bx_e \in \textbf{BExp} \cup \{e\} \), and 
\( a_e \in \textbf{Assign} \cup \{e\} \)
Proof: Proof is the same as for (3)2.

(3)4. Q.E.D.
Proof: By (3)1 - (3)3 and definition of \( \textbf{Act} \) (Eq. (7)) and (Eq. (8)).

(2)2. Q.E.D.
Proof: By (2)1 and logical implication.

(1)2. Case: 1.1 \( n = n' + 1 \)
(1)2. \( \forall j \in \mathbb{N} \setminus \{0\} : j \leq n' \implies \text{Ind}((\text{act}_1, \ldots, \text{act}_n), j), EL \)
Proof: By (2)1 and logical implication.
Proof: By assumption 2.1, assumption 2.3, and definition of A (Eq. (10)).

(3.2) Choose \( aas \in A \) and \( as_1 \in \{act_1\}, \ldots, as_{n-1} \in \{act_{n-1}\} \) such that
\[
as = as_1 \cup \cdot \cdot \cdot \cup as_{n-1} \text{ and } \forall i \in \{1, \ldots, n-1\} : \text{preState}(as_i, \text{var}(EL)) = \text{preState}(as'_i, \text{var}(EL)) / \text{postState}(as_i, \text{var}(EL)) = \text{postState}(as'_i, \text{var}(EL))
\]
Proof: By (3.1) and assumptions 1.1, 1.2, 2.2, and 2.3.

(3.3) Choose \( as_n \in \{act_n\} \) such that \( aas \cup as_n \in A, \text{preState}(as_n, \text{var}(EL)) = \text{postState}(aas, \text{var}(EL)) = \text{postState}(as_n, \text{var}(EL)) = \text{postState}(as'_n, \text{var}(EL)) \)
Proof: By (3.2) and definition of \( \bot \) (Def. A.3).

(4.3) \( \text{postState}(as_n, \text{var}(EL)) = \text{postState}(as'_n, \text{var}(EL)) \)
Proof: By assumption 2.1, assumption 2.3, and definition of \( \bot \) (Def. A.3).

(4.4) \( \text{postState}(as_n, \text{var}(EL)) = \text{postState}(as'_n, \text{var}(EL)) \)
Proof: By (4.1), (4.2), and (4.3).

(5.5) CASE: 3.1 \( act_n = (i, bx, \epsilon, a) \) for \( bx \in BExp \cup \{\epsilon\} \) and \( a \in Assign \cup \{\epsilon\} \)

(6.6) CASE: 4.1 \( ie \in Dm(\text{te}_{\text{EL}}) \)

(7.7) \( \text{Choose } aas'i \in \{(i, bx, \epsilon, \epsilon)\}, aas' \in \{(\epsilon, bx, \epsilon, \epsilon)\}, \) and \( aas' \) such that \( aas' = aas'i \cup aas' \)
Proof: By assumption 2.2, assumption 3.1, assumption 4.1, and definition of \( T'_{\text{EL}} \) (Def. B.3).

(8.8) \( \text{postState}(as_n, \text{var}(EL)) = \text{postState}(as'_n, \text{var}(EL)) \)
Proof: By assumption 1, assumption 3.1, (4.1), (4.2), (7.1), and definition of \( \bot \) (Def. A.3).

(9.9) Q.E.D.
Proof: By (7.1) and (7.2).

(10.10) CASE: 4.1 \( ie \notin Dm(\text{te}_{\text{EL}}) \)

(11.11) \( \text{Choose } as'_n \in \{act_n\} \) \( \{(\epsilon, bx, \epsilon, \epsilon)\} \)
Proof: By assumption 2.2, assumption 3.1, and definition of \( T'_{\text{EL}} \) (Def. B.3).

(12.12) Q.E.D.
Proof: By assumption 1, (4.1), (4.2), (7.1), and definition of \( \bot \) (Def. A.3).

(13.13) Q.E.D.
Proof: By (6.1) and (6.2).

(14.14) CASE: 3.1 \( act_n = (i, bx, oe, a) \) for \( oe \in OE, bx \in BExp \cup \{\epsilon\} \), and \( a \in Assign \cup \{\epsilon\} \)
Proof: Proof is the same as for (5)2.

(5)4. Q.E.D.

Proof: By (5)1 - (5)3 and definition of Act (Eq. (7)) and (Eq. (8)).

⟨4⟩4. aas^n \in A

Proof: By assumption 2.1 - 2.3, ⟨3⟩2, ⟨4⟩1, ⟨4⟩2, assumption 1, definition of T_EL (Def. B.3), and definition of A (Def. 10).

(4)5. Q.E.D.

Proof: By ⟨4⟩1 - ⟨4⟩4.

⟨3⟩4. Q.E.D.

Proof: By ⟨3⟩1, ⟨3⟩2 and ⟨3⟩3.

⟨2⟩2. Q.E.D.

Proof: By ⟨2⟩1 and logical implication.

⟨1⟩3. Q.E.D.

Proof: By ⟨1⟩1 and ⟨1⟩2.

Lemma T.B.1.3 If

- EL_1 = rnm(P, EL) and EL_2 = rnm(Q, EL) for some P, Q ∈ P and EL, EL_1, EL_2 ∈ EL

- (act_1, ..., act_1,j) ∈ pr(|P|)

- as_1 \in \langle T'_{EL_1}(act_1.1) \rangle \land \cdots \land as_1 \in \langle T'_{EL_1}(act_1.j) \rangle

- as_1 \in A and as_1 = as_1 \sim \cdots \sim as_1

- as_1 \in \langle act_1.1 \rangle \land \cdots \land as_1 \in \langle act_1.j \rangle

- \forall i \in \{1, ..., j\} : (preState(as_1.i, var(EL_1)) = preState(as_1.i, var(EL_1)) \land
  (postState(as_1.i, var(EL_1)) = postState(as_1.i, var(EL_1)))

- as_1 \in pr(|P|) \cap A and as_1 = as_1 \sim \cdots \sim as_1

- as_2 \in pr(|Q|) \cap A and io(as_2) = io(as_1)

- (act_2, ..., act_2.k) ∈ pr(|Q|)

- as_2 \in \langle act_2.1 \rangle, ..., as_2 \in \langle act_2.k \rangle

- as_2 = as_2 \sim \cdots \sim as_2

then

- \exists as_2 \in \langle T'_{EL_2}(act_2.1) \rangle \sim \cdots \sim \langle T'_{EL_2}(act_2.k) \rangle : as_2 = as_2 \in A \land io(as_2) = io(as_1)

Proof of Lemma T.B.1.3 The proof is by induction on the sum j and k. In the proof, we make use of the following definition which high-lights the
\[
\text{Proofs}
\]

\[
\begin{align*}
\text{Ind}(P, Q, EL_1, EL_2, n) & \overset{\text{def}}{=} \\
\forall j, k \in \mathbb{N} \setminus \{0\} : \\
\forall (act_{1,1}, \ldots, act_{1,j}) \in pr(|P|) : \\
\forall s_{1,1} ' \in \{T'_{EL_1}(act_{1,1})\} : \ldots \forall s_{1,j} ' \in \{T'_{EL_1}(act_{1,j})\} : \\
\forall s_{1,1} ' \in \{act_{1,1}\} : \ldots \forall s_{1,j} ' \in \{act_{1,j}\} : \\
\forall (act_{2,1}, \ldots, act_{2,k}) \in pr(|Q|) : \\
\forall s_{2,1} ' \in \{act_{2,1}\} : \ldots \forall s_{2,k} ' \in \{act_{2,k}\} : \\
\forall s_j ' \in A : \forall s_1 ' \in pr(|P|) \cap A : \forall s_2 ' \in pr(|Q|) \cap A : \\
\wedge j + k = n \\
\wedge as_1 ' = as_{1,1} ' \ldots as_{1,j} ' \\
\wedge as_1 = as_{1,1} \ldots as_{1,j} \\
\wedge as_2 = as_{2,1} \ldots as_{2,k} ' \\
\wedge io(as_1) = io(as_2) \\
\wedge \forall i \in \{1, \ldots, j\} : \\
\wedge \text{preState}(as_{1,i}, \text{var}(EL_1)) = \text{preState}(as_{1,i}', \text{var}(EL_1)) \\
\wedge \text{postState}(as_{1,i}, \text{var}(EL_1)) = \text{postState}(as_{1,i}', \text{var}(EL_1)) \\
\Rightarrow \exists s_k ' \in \{T'_{EL_2}(act_{2,1})\} \ldots \{T'_{EL_2}(act_{2,k})\} : \\
\wedge as_k ' \in A \\
\wedge io(as_k') = io(as_1')
\end{align*}
\]

**Assume:** 1. $EL_1 = \text{rm}(P, EL)$ and $EL_2 = \text{rm}(Q, EL)$ for some $P, Q \in P$ and $EL, EL_1, EL_2 \in EL$
2. $n \geq 2$ for some $n \in \mathbb{N}$

**Prove:** $\text{Ind}(P, Q, EL_1, EL_2, n)$

\(1\). **Case:** 1.1 $n = 2$

\(\langle 2 \rangle \). **Assume:** 2.1 $j = 1$ and $k = 1$ for $j, k \in \mathbb{N} \setminus \{0\}$

\(\langle 2.2 \rangle \). $\langle act_1 \rangle \in pr(|P|)$

\(\langle 2.3 \rangle \). $s_1 ' \in \{T'_{EL_1}(act_1)\}$

\(\langle 2.4 \rangle \). $s_1 ' \in A$

\(\langle 2.5 \rangle \). $as_1 ' \in \{act_1\}$

\(\langle 2.6 \rangle \). $(\text{preState}(as_1, \text{var}(EL_1)) = \text{preState}(as_1 ', \text{var}(EL_1))) \wedge (\text{postState}(as_1, \text{var}(EL_1)) = \text{postState}(as_1 ', \text{var}(EL_1)))$

\(\langle 2.7 \rangle \). $as_1 ' \in pr(|P|) \cap A$

\(\langle 2.8 \rangle \). $as_2 ' \in pr(|Q|) \cap A$ and $io(as_2) = io(as_1)$

\(\langle 2.9 \rangle \). $\langle act_2 \rangle \in pr(|Q|)$

\(\langle 2.10 \rangle \). $as_2 ' \in \{act_2\}$

**Prove:** $\exists s_2 ' \in \{T'_{EL_2}(act_2)\} : s_2 ' \in A \land io(s_2') = io(s_1')$

\(\langle 3 \rangle \). **Case:** 3.1 $act_1 = (\epsilon, bx_\epsilon, \epsilon, a_\epsilon)$ for $bx_\epsilon \in BExp \cup \{\epsilon\}$ and $a_\epsilon \in \text{Assign} \cup \{\epsilon\}$

\(\langle 4 \rangle \). **Case:** 4.1 $act_2 = (\epsilon, bx_\epsilon', \epsilon, a_\epsilon')$ for $bx_\epsilon ' \in BExp \cup \{\epsilon\}$ and $a_\epsilon ' \in \text{Assign} \cup \{\epsilon\}$

\(\langle 5 \rangle \). **Choose** $s_2 ' \in \{T'_{EL_2}(act_2)\}$ such that $s_2 ' = s_2$

\(\langle 6.1 \rangle \). $\{T'_{EL_1}(act_1)\} = \{\text{tact}_{EL_2}(act_2)\} = \{act_2\}$

**Proof:** By definition of $T'_{EL_2}$ (Def. B.3), definition of $\text{tact}_{EL}$ (Eq. (15)), and assumption 4.1.

\(\langle 6.2 \rangle \). Q.E.D.

**Proof:** By \(\langle 6.1 \rangle \) and assumption 2.10.
(5.2) \( as'_2 \in A \)

Proof: By (5.1) and assumption 2.8.

(5.3) \( \text{io}(as'_2) = \text{io}(as'_1) \)

(6.1) \( \text{io}(as'_2) = \emptyset \)

Proof: By (5.1), assumption 2.10, assumption 4.1, definition of \( \llparenthesis \) (Def. A.3) and definition of \( \text{io} \) (Eq. (11)).

(6.2) \( \text{io}(as'_1) = \emptyset \)

Proof: By assumption 2.3, assumption 3.1, definition of \( T'_{EL_1} \), (Def. B.3), definition of \( \llparenthesis \) (Def. A.3) and definition of \( \text{io} \) (Eq. (11)).

(6.3) Q.E.D.

Proof: By (6.1) and (6.2).

(5.4) Q.E.D.

Proof: By (5.1) - (5.3).

(4.2) Case: 4.1 \( act_2 = (ie', bx'_r, \epsilon, a'_r) \) for \( ie' \in IE \cup \{ \epsilon \}, bx'_r \in \text{BExp} \cup \{ \epsilon \}, \) and \( a'_r \in \text{Assign} \cup \{ \epsilon \} \)

Proof: Case assumption 4.1 contradicts assumption 2.8 since assumption 4.1 implies that \( \text{io}(as'_2) \neq \text{io}(as'_1) \) by definition of \( \llparenthesis \) (Def. A.3) and definition of \( \text{io} \) (Eq. (11)). Hence, assumption 4.1 cannot hold.

(4.3) Case: 4.1 \( act_2 = (\epsilon, bx'_r, \epsilon, a'_r) \) for \( \epsilon \in OE \cup \{ \epsilon \}, bx'_r \in \text{BExp} \cup \{ \epsilon \}, \) and \( a'_r \in \text{Assign} \cup \{ \epsilon \} \)

Proof: Case assumption 4.1 contradicts assumption 2.8 since assumption 4.1 implies that \( \text{io}(as'_2) \neq \text{io}(as'_1) \) by definition of \( \llparenthesis \) (Def. A.3) and definition of \( \text{io} \) (Eq. (11)). Hence, assumption 4.1 cannot hold.

(4.4) Q.E.D.

Proof: By (4.1) - (4.3) and definition of \( \text{Act} \) (Eq. (7)) and (Eq. (8)).

(3.2) Case: 3.1 \( act_1 = (ie, bx_r, \epsilon, a_r) \) for \( ie \in IE \cup \{ \epsilon \}, bx_r \in \text{BExp} \cup \{ \epsilon \}, \) and \( a_r \in \text{Assign} \cup \{ \epsilon \} \)

(4.1) Case: 4.1 \( act_2 = (\epsilon, bx'_r, \epsilon, a'_r) \)

Proof: Case assumption 4.1 contradicts assumption 2.8 since assumption 4.1 implies that \( \text{io}(as'_2) \neq \text{io}(as'_1) \) by definition of \( \llparenthesis \) (Def. A.3) and definition of \( \text{io} \) (Eq. (11)). Hence, assumption 4.1 cannot hold.

(4.2) Case: 4.1 \( act_2 = (ie', bx'_r, \epsilon, a'_r) \) for \( ie' \in IE \cup \{ \epsilon \}, bx'_r \in \text{BExp} \cup \{ \epsilon \}, \) and \( a'_r \in \text{Assign} \cup \{ \epsilon \} \)

(5.1) Case: 5.1 \( ie \in \text{Dom}(te_{EL_1}) \) and \( ie' \in \text{Dom}(te_{EL_2}) \)

(6.1) \( \exists as'_2 \in \llparenthesis (ie, bx'_r, \epsilon, a_r) \cdot te_{EL_2}(ie') \cdot (\epsilon, \epsilon, \epsilon, a'_r) \rrparenthesis : as'_2 \in A \land \text{io}(as'_1) = \text{io}(as'_2) \)

(7.1) \( \exists as'_2 \in \llparenthesis (\epsilon, bx'_r, \epsilon, \epsilon) \rrparenthesis \land \llparenthesis (te_{EL_2}(ie')) \rrparenthesis \land \llparenthesis (\epsilon, \epsilon, \epsilon, a'_r) \rrparenthesis : as'_2 \in A \land \text{io}(as'_1) = \text{io}(as'_2) \)

(8.1) Choose \( as'_{1,1} \in \llparenthesis (\epsilon, bx_r, \epsilon, \epsilon) \rrparenthesis, as'_{1,2} \in \llparenthesis te_{EL_1}(ie) \rrparenthesis, \) and \( as'_{1,3} \in \llparenthesis (\epsilon, \epsilon, \epsilon, a_r) \rrparenthesis \) such that \( as'_1 = as'_{1,1} \land as'_{1,2} \land as'_{1,3} \)

Proof: By definition of \( T_{EL_1} \) (Def. B.3), assumption 3.1, assumption 5.1, and definition of \( \text{tact}_{EL} \) (Eq. (15)).

(9.1) \( \llparenthesis T'_{EL_1}(act_1) \rrparenthesis = \llparenthesis \text{tact}_{EL_1}(act_1) \rrparenthesis = \llparenthesis (ie, bx_r, \epsilon, \epsilon) \cdot te_{EL_1}(ie) \cdot (\epsilon, \epsilon, \epsilon, a_r) \rrparenthesis \)

Proof: By definition of \( T'_{EL_1} \) (Def. B.3), assumption 3.1.

(9.2) \( \llparenthesis (\epsilon, bx_r, \epsilon, \epsilon) \cdot te_{EL_1}(ie) \cdot (\epsilon, \epsilon, \epsilon, a_r) \rrparenthesis = \llparenthesis (\epsilon, bx_r, \epsilon, \epsilon) \rrparenthesis \land \llparenthesis te_{EL_1}(ie) \rrparenthesis \land \llparenthesis (\epsilon, \epsilon, \epsilon, a_r) \rrparenthesis \)

Proof: By definition of \( \llparenthesis \) (Def. A.3).

(9.3) Q.E.D.

Proof: By (9.1), (9.2), and assumption 2.3.

(8.2) Choose variable renaming functions \( vr_1, vr_2 \in \text{Var} \) such that \( vr_1(EL) = EL_2 \) and \( vr_2(EL) = EL_2 \) when
vr_1 and vr_2 are lifted to event libraries.

**Proof:** By assumption 1.

(8.3) Choose as'_{2,1} \in \langle (e, bx', e, e) \rangle such that \text{preState}(as'_{2,1}, \text{var}(EL_2)) = \text{preState}(as_2, \text{var}(EL_2)) and \text{preState}(as'_{2,1})(vr_2(x)) = \text{preState}(as'_{1,1})(vr_1(x)) for all x \in \text{var}(EL)

**Proof:** By assumptions 1, 2.9, and 4.1, and (8.1), (8.2), definition of \(\langle \_ \rangle \) (Def. A.3), \text{preState}(\_\_\_), and \text{preState}(\_\_\_) (Def. D.2).

(8.4) as'_{2,1} \in A

**Proof:** By (8.3) and assumption 2.8, and definition of A (Eq. (10)).

(8.5) io(as'_{1,1}) = io(as'_{2,1}) = \langle \rangle

**Proof:** By (8.1), (8.3) and definition of \(\langle \_ \rangle \) (Def. A.3) and io (Eq. (11)).

(8.6) Choose as'_{2,2} \in \langle te_{EL_2}(ie') \rangle such that \text{preState}(as'_{2,2}) = \text{postState}(as'_{1,1})

**Proof:** By (8.3), assumption 5.1, definition of \(\langle \_ \rangle \) (Def. A.3), and definition of preState(\_\_\_) and postState(\_\_\_) (Def. D.2).

(8.7) as'_{2,2} \in A

**Proof:** By assumption 2.4, assumption 2.6, (8.1), (8.3), (8.6), and definition of A (Eq. (10)).

(8.8) io(as'_{2,2}) = io(as'_{1,2})

**Proof:** By assumption 2.6, assumption 2.8, (8.1), (8.3), (8.6), and definition of io (Eq. (11)).

(8.9) Choose as'_{2,3} \in \langle (e, e, e, a'_2) \rangle such that \text{preState}(as'_{2,3}) = \text{postState}(as'_{2,2})

**Proof:** By definition of \(\langle \_ \rangle \) (Def. A.3), and definition of preState() and postState() (Def. D.2).

(8.10) as'_{2,3} \in A

**Proof:** By (8.9) and definition of A (Eq. (10)).

(8.11) io(as'_{2,3}) = io(as'_{1,3})

**Proof:** By (8.1), (8.9), definition of \(\langle \_ \rangle \) (Def. A.3), and definition of io (Eq. (11)).

(8.12) Choose as'_{2} \in A such that as'_{2} = as'_{2,1} \sim as'_{2,2} \sim as'_{2,3}

**Proof:** By (8.4), (8.6), (8.7), (8.9), (8.10), and definition of A (Eq. (10)).

(8.13) io(as'_{1}) = io(as'_{2})

**Proof:** By (8.1), (8.5), (8.8), (8.11), (8.12), and definition of io (Eq. (11)).

(8.14) Q.E.D.

**Proof:** By (8.3), (8.6), (8.9), (8.12) and (8.13).

(7.2) Q.E.D.

**Proof:** By (7.1) and definition \(\langle \_ \rangle \) (Def. A.3).

(6.2) Q.E.D.

**Proof:** By assumption 5.1, (6.1), and definition of T'_{EL_2} (Def. B.3).

(5.2) CASE: 5.1 ie /\in \text{Dom}(te_{EL_1}) and ie' /\in \text{Dom}(te_{EL_2})

(6.1) Choose as'_{2} \in \langle T'_{EL_2}(act_2) \rangle such that as'_{2} = as_2

(7.1) \{T'_{EL_2}(act_2)\} = \{\text{tact}_{EL_2}(act_2)\} \neq \{act_2\}

**Proof:** By assumption 4.1, assumption 5.1, and definition of T'_{EL_2} (Def. B.3), definition \(\langle \_ \rangle \) (Def. A.3), and definition of...
\[ t_{act_{EL}} \text{ (Eq. (15)}. \]

(7) 2. Q.E.D.

Proof: By (7) 1 and assumption 2.10.

(6) 2. \( as'_2 \in A \)

Proof: By (6) 1 and assumption 2.8.

(6) 3. \( a_1 = as'_1 \)

(7) 1. \( \langle T_{act_1} \rangle = \langle t_{act_{EL_1}}(act_1) \rangle = \langle act_1 \rangle \)

Proof: By assumption 3.1, assumption 5.1, and definition of \( T'_{EL_1} \) (Def. B.3), definition (\ref{eq:6}) (Def. A.3), and definition of \( t_{act_{EL_1}} \) (Eq. (15)).

(7) 2. Q.E.D.

Proof: By (7) 1, assumptions 2.3 and 2.5.

(6) 4. \( io(as'_2) = io(as'_1) \)

Proof: By (6) 1, (6) 2, and assumption 2.8.

(6) 5. Q.E.D.

Proof: By (6) 1, (6) 2, and (6) 4.

(5) 3. Q.E.D.

Proof: By (5) 1 and (5) 2.

(4) 3. Case 2: \( 4.1 \ act_2 = (\epsilon, bx'_1, \epsilon, a'_1) \) for oe' \( \in OE \cup \{\epsilon\}, bx'_1 \in BExp \cup \{\epsilon\}, \) and \( a'_1 \in Assign \cup \{\epsilon\} \)

Proof: Case assumption 4.1 contradicts assumption 2.8 since assumption 4.1 implies that \( io(as_2) \neq io(as_1) \) by definition of \( \{\epsilon\} \) (Def. A.3) and definition of \( io \) (Eq. (11)). Hence, assumption 4.1 cannot hold.

(4) 4. Q.E.D.

Proof: By (4) 1, (4) 2, (4) 3 and definition of \( Act \) (Eq. 7) and (Eq. 8).

(3) 3. Case 2: \( 3.1 \ act_1 = (\epsilon, bx, \epsilon, a, a'_1) \) for oe \( \in OE \cup \{\epsilon\}, bx \in BExp \cup \{\epsilon\}, \) and \( a_1 \in Assign \cup \{\epsilon\} \)

Proof: Proof is the same as for (3) 2.

(3) 4. Q.E.D.

Proof: By (3) 1, (3) 2, (3) 3 and definition of \( Act \) (Eq. 7) and (Eq. 8).

(2) 2. Q.E.D.

Proof: By (2) 1 and logical implication (\( \implies \)).

(1) 2. Case 1: \( 1.1 \ n = n' + 1 \) for \( n' \in \mathbb{N} \)

\( 1.2 \ \forall i \in \mathbb{N} : i \geq 2 \land i \leq n' \implies \text{Ind}(P, Q, EL_1, EL_2, i) \)

(2) 1. Assume: \( 2.1 \ j + k = n \) for \( j, k \in \mathbb{N} \setminus \{0\} \)

\( 2.2 \ \langle act_{1,1}, \ldots, act_{1,j} \rangle \in pr(|P|) \)

\( 2.3 \ as'_{1,1} \in \{T_{act_1,1,1} \} \land \cdots \land as'_{1,j} \in \{T_{act_1,1,j} \} \)

\( 2.4 \ as'_1 \in A \) and \( as'_1 = as'_{1,1} \cdots as'_{1,j} \)

\( 2.5 \ \forall i \in \{1, \ldots, j\} : \text{preState}(as_{1,i}, var(EL_1)) = \text{preState}(as'_1, var(EL_1)) \land \text{postState}(as_{1,i}, var(EL_1)) = \text{postState}(as'_1, var(EL_1)) \)

\( 2.7 \ as_1 \in pr(|P|) \cap A \) and \( as_1 = as'_{1,1} \cdots as_1,j \)

\( 2.8 \ as_2 \in pr(|Q|) \cap A \) such that \( io(as_2) = io(as_1) \)

\( 2.9 \ \langle act_{2,1}, \ldots, act_{2,k} \rangle \in |Q| \)

\( 2.10 \ as_{2,1} \in \{act_{2,1} \} \land \cdots \land as_{2,k} \in \{act_{2,k} \} \)

\( 2.11 \ as_2 = as_{2,1} \cdots as_{2,k} \)

Proof: \( \exists as'_2 \in \{T_{act_{EL_2}}(act_2) \} : as'_2 \in A \land io(as'_2) = io(as'_1) \)

(3) 1. Case 3: \( 3.1 \ act_{1,j} = (\epsilon, bx, \epsilon, a_1) \) for \( bx \in BExp \cup \{\epsilon\} \) and \( a_1 \in Assign \cup \{\epsilon\} \)
(4)1. Choose $aas_1' \in A$ such that $as_1' = aas_1' \sim as_{1,j}$.  
Proof: By assumption 2.4 and definition of $A$ (Eq. (10)).

(4)2. Choose $aas_1 \in pr(\langle P \rangle) \cap A$ such that $as_1 = aas_1 \sim as_{1,j}$.  
Proof: By assumption 2.7, definition of $A$ (Eq. (10)), and definition of $pr(\_)$ (Def. D.3).

(4)3. $io(aas_1) = io(aas_2)$  
Proof: By (4)2, assumptions 2.5, 2.8 and 3.1, and definition of $\langle \_ \rangle$ (Def. A.3) and definition of $io$ (Eq. (11)).

(4)4. Q.E.D.

Proof: By assumptions 1.1, 1.2, 2.1, 2.2, 2.3, 2.5, 2.6, and 2.8 - 2.11, and (4)1, (4)2, and (4)3 since this implies that the antecedent of $Ind(P, Q, EL_1, EL_2, n')$ is satisfied for $j - 1$ and $k$.

(3)2. Case: 3.1 $act_{1,j} = (ie, bx_\epsilon, \epsilon, a_\epsilon)$ for $ie \in IE$, $bx_\epsilon \in BExp \cup \{\epsilon\}$, and $a_\epsilon \in Assign \cup \{\epsilon\}$

(4)1. Case: 4.1 $act_{2,k} = (\epsilon, bx_\epsilon', \epsilon, a_\epsilon')$ for $bx_\epsilon' \in BExp \cup \{\epsilon\}$ and $a_\epsilon' \in Assign \cup \{\epsilon\}$

(5)1. Choose $aas_2 \in pr(\langle Q \rangle) \cap A$ such that $as_2 = aas_2 \sim as_{2,k}$.  
Proof: By assumptions 2.8 and 2.11, and definition of $A$ (Eq. (10)), and definition of $pr(\_)$ (Def. D.3).

(5)2. $io(aas_1) = io(aas_2)$  
Proof: By (5)1, assumptions 4.1 and 2.8, and definition of $\langle \_ \rangle$ (Def. A.3) and definition of $io$ (Eq. (11)).

(5)3. Choose $aas_2' \in \langle T'_{EL_2}(act_{2,1}) \rangle \cap A$ such that $io(aas_2') = \langle \_ \_ \rangle$ and $preState(aas_2') = postState(aas_2')$.  
Proof: By (5)2, assumption 4.1, assumption 1, assumption 2.8 - 2.11, definition of $T'_{EL_2}$ (Def. B.3), and definition of $\langle \_ \_ \rangle$ (Def. A.3).

(5)4. $aas_2' \sim aas_{2,k}' \in A$ and $io(aas_2' \sim aas_{2,k}') = io(aas_1')$  
Proof: By (5)3, (5)4, definition of $A$ (Eq. (10)), and definition of $io$ (Eq. (11)).

(5)5. Q.E.D.

Proof: By (5)3, (5)4, and (5)5.

(4)2. Case: 4.1 $act_{2,k} = (ie', bx_\epsilon', \epsilon, a_\epsilon')$ for $ie' \in IE$, $bx_\epsilon' \in BExp \cup \{\epsilon\}$, and $a_\epsilon' \in Assign \cup \{\epsilon\}$

(5)1. Choose $aas_1' \in A$ such that $as_1' = aas_1' \sim as_{1,j}$  
Proof: By assumption 2.4 and definition of $A$ (Eq. (10)).

(5)2. Choose $aas_1 \in pr(\langle P \rangle) \cap A$ such that $as_1 = aas_1 \sim as_{1,j}$.  
Proof: By assumption 2.7, definition of $A$ (Eq. (10)), and definition of $pr(\_)$ (Def. D.3).

(5)3. Choose $aas_2 \in pr(\langle Q \rangle) \cap A$ such that $as_2 = aas_2 \sim as_{2,k}$.  
Proof: By assumptions 2.8 and 2.11, and definition of $A$ (Eq. (10)), and definition of $pr(\_)$ (Def. D.3).

(5)4. $io(aas_1) = io(aas_2)$  
Proof: By (5)2, (5)3, assumptions 2.8, 3.1, and 4.1, and definition of $io$ (Eq. (11)).

(5)5. Choose $aas_2' \in \langle T'_{EL_2}(act_{2,1}) \rangle \cap A$ such
that \(io(aas'_1) = io(aas'_2)\)

**Proof:** By assumptions 1.1, 1.2, 2.1, 2.2, 2.3, 2.5, 2.6, 2.9, 2.10, 2.11, and (5)1 - (5)4 since this implies that the antecedent of \(Ind(P, Q, EL_1, EL_2, n'-1)\) is satisfied for \(j - 1\) and \(k - 1\).

(5)6. Choose \(as'_2 \in \{T'_{EL_2} act_{2,k} \} \cap A\) such that \(io(as'_{2,k}) = io(as'_{1,j})\)

and \(postState(as'_{2,k}) = postState(as'_{2,k})\)

(6)1. Case: 5.1 \(ie \in Dom(te_{EL_1})\) and \(ie' \in Dom(te_{EL_2})\)

(7)1. \(\exists as'_{2,k} \in \{ (e, bx', e, e, te_{EL_2}(ie'), (e, e, e, a'_1)) \} \cap A \land io(as'_{2,k}) = io(as'_{1,j}) \land postState(as'_{2,k}) = postState(as'_{2,k})\)

(8)1. \(\exists as'_{2,k} \in \{ (e, bx', e, e) \} \cap \{ te_{EL_2}(ie') \} \cap \{ (e, e, e, a'_1) \} \cap A \land io(as'_{2,k}) = io(as'_{1,j}) \land postState(as'_{2,k}) = postState(as'_{2,k})\)

(9)1. Choose \(aas'_{1,1} \in \{ (e, bx, e, e) \}, aas'_{1,2} \in \{ te_{EL_1}(ie) \}, aas'_{1,3} \in \{ (e, e, e, a) \}\) such that \(aas'_{1,2} - aas'_{1,3}\)

(10)1. \(\{ T'_{EL_1} act_{1,j} \} = \{ tact_{EL_1}(act_{1,j}) \} = \{ (e, bx, e, e, te_{EL_1}(ie), (e, e, e, a)) \}\)

**Proof:** By definition of \(T'_{EL_1}\) (Def. B.3), assumption 3.1, assumption 5.1, and definition of \(tact_{EL}\) (Eq. (15)).

(10)2. \(\{ (e, bx, e, e, te_{EL_1}(ie), (e, e, e, a)) \} = \{ (e, bx, e, e) \} \land \{ te_{EL_1}(ie) \} \land \{ (e, e, e, a) \}\)

**Proof:** By definition of \(\llcorner \cdot \lrcorner\) (Def. A.3).

(10)3. Q.E.D.

**Proof:** By (10)1, (10)2, and assumption 2.3.

(9)2. Choose variable renaming functions \(vr_1, vr_2 \in Var \rightarrow Var\) such that \(vr_1(EL) = EL_2\) and \(vr_2(EL) = EL_2\) when \(vr_1\) and \(vr_2\) are lifted to event libraries.

**Proof:** By assumption 1.

(9)3. Choose \(aas'_{2,1} \in \{ (e, bx', e, e) \}\) such that \(preState(aas'_{2,1}) =\)

\(postState(aas'_{2,1})\) and \(preState(aas'_{2,1})(vr_2(x)) = postState(aas'_{1,1})(vr_1(x))\)

for all \(x \in var(EL)\)

**Proof:** By assumptions 1, 2.9, and 4.1, and (5)5, (9)1, definition of \(\llcorner \cdot \lrcorner\) (Def. A.3), and definition of \(postState()\) (Def. D.2).

(9)4. \(aas'_{2,1} \in A\)

**Proof:** By (9)3, assumption 2.8, assumption 2.10, assumption 4.1, and definition of \(A\) (Eq. (10)).

(9)5. \(io(aas'_{1,1}) = io(aas'_{2,1})\)

**Proof:** By (9)1, (9)3, and definition of \(\llcorner \cdot \lrcorner\) (Def. A.3) and \(io\) (Eq. (11)).

(9)6. Choose \(aas'_{2,2} \in \{ te_{EL_2}(ie') \}\) such that \(preState(aas'_{2,2}) =\)

\(postState(aas'_{2,1})\)

**Proof:** By (9)3, assumption 5.1, definition of \(\llcorner \cdot \lrcorner\) (Def. A.3), and definition of \(preState()\) and \(postState()\) (Def. D.2).

(9)7. \(aas'_{2,2} \in A\)

**Proof:** By assumption 2.4, assumption 2.6, (9)1, (9)3, (9)6, and definition of \(A\) (Eq. (10)).

(9)8. \(io(aas'_{2,2}) = io(aas'_{1,2})\)

**Proof:** By assumption 2.6, assumption 2.8, (9)1, (9)3, (9)6, and definition of \(io\) (Eq. (11)).

(9)9. Choose \(aas'_{2,3} \in \{ (e, e, e, a') \}\) such that \(preState(aas'_{2,3}) =\)

\(postState(aas'_{2,2})\)
\textbf{PROOFS}

\section*{Proofs}

Proof: By definition of \( \downarrow \) (Def. A.3), and definition of \textit{preState}() and \textit{postState}() (Def. D.2).

\smallskip

(9.10) \textit{aas}_{2,3} \in A

\textbf{Proof:} By \{9\}9 and definition of \( A \) (Eq. (10)).

(9.11) \textit{io}(aas'_{2,3}) = \textit{io}(aas'_{1,3})

\textbf{Proof:} By \{9\}1, \{9\}9, definition of \( \downarrow \) (Def. A.3), and definition of \textit{io} (Eq. (11)).

(9.12) Choose \( aas'_2 \in A \) such that \( aas'_2 = aas'_{2,1} \sim aas'_{2,2} \sim aas'_{2,3} \)

\textbf{Proof:} By \{9\}4, \{9\}6, \{9\}7, \{9\}9, \{9\}10, and definition of \( A \) (Eq. (10)).

(9.13) \textit{io}(aas'_{1,j}) = \textit{io}(aas'_2)

\textbf{Proof:} By \{9\}1, \{9\}5, \{9\}8, \{9\}11, \{9\}13, and definition of \textit{io} (Eq. (11)).

(9.14) Q.E.D.

\textbf{Proof:} By \{9\}12, \{9\}13.

(8.2) Q.E.D.

\textbf{Proof:} By \{8\}1 and definition \( \downarrow \) (Def. A.3).

(7.2) Q.E.D.

\textbf{Proof:} By assumption 5.1, \{7\}1, and definition of \( T'_{EL_2} \) (Def. B.3).

(6.2) Case: \( ie \notin \text{Dom}(te_{EL_1}) \) and \( ie' \notin \text{Dom}(te_{EL_2}) \)

\textbf{Proof:} By \{7\}1 and assumption 2.8.

(7.1) Choose \( as'_{2,k} \in T'_{EL_2}(act_{2,k}) \) such that \( as'_{2,k} = as_{2,k} \)

\textbf{Proof:} By assumption 4.1, assumption 5.1, and definition of \( T'_{EL_2} \) (Def. B.3), and definition of \textit{tact}_{EL_2} (Eq. (15)).

(8.2) Q.E.D.

\textbf{Proof:} By \{8\}1 and assumption 2.10.

(7.2) \textit{as}'_{2,k} \in A

\textbf{Proof:} By \{7\}1 and assumption 2.8.

(7.3) \textit{as}'_{1,j} \in \textit{as}_{1,j}

\textbf{Proof:} By \{7\}1, \{7\}3, and assumption 2.8.

(7.4) \textit{io}(as'_{2,k}) = \textit{io}(as'_{1,j})

\textbf{Proof:} By \{7\}1, \{7\}2, and \{7\}4.

(6.3) Case: \( ie \notin \text{Dom}(te_{EL_1}) \) and \( ie' \notin \text{Dom}(te_{EL_2}) \)

\textbf{Proof:} Case assumption 4.1 contradicts assumption 2.8 since assumption 4.1 implies that \( \textit{io}(as_2) \neq \textit{io}(as_1) \) by assumption 1, definition of \textit{rmn} (App.B.1), definition of \textit{io} (Eq. (11)) and definition of \textit{te}_{EL} (Eq. 14). Hence, the case assumption cannot hold.

(6.4) Case: \( ie \in \text{Dom}(te_{EL_1}) \) and \( ie' \notin \text{Dom}(te_{EL_2}) \)

\textbf{Proof:} Case assumption 4.1 contradicts assumption 2.8 since assumption 4.1 implies that \( \textit{io}(as_2) \neq \textit{io}(as_1) \) by assumption 1, definition of \textit{rmn} (App.B.1), definition of \textit{io} (Eq. (11)) and definition of \textit{te}_{EL} (Eq. 14). Hence, the case assumption cannot hold.

(6.5) Q.E.D.
Lemma B.1 If \( EL \) is an event library, and \( P_1 \) and \( P_2 \) be basic state machines whose variables are disjoint from those in \( EL \), then \( T'_{EL} \) is homomorphic w.r.t. union, i.e.,

\[
[T'_{EL}([P_1]) \cup [P_2]) = [T'_{EL}([P_1])] \cup [T'_{EL}([P_2])]
\]

Proof of Lemma B.1.1

\[
\begin{align*}
[T'_{EL}(P_1) + T'_{EL}(P_2)] &= [T'_{EL}(P_1) + T'_{EL}(P_2)] \\
[T'_{EL}(P_1 + P_2)] &= [T'_{EL}(P_1)] \cup [T'_{EL}(P_2)] \\
[T'_{EL}([P_1] \cup [P_2])] &= [T'_{EL}([P_1])] \cup [T'_{EL}([P_2])]
\end{align*}
\]

Lemma B.1.2

\[ [P + Q] = [P] \cup [Q] \]

Proof of Lemma B.1.2

\[
\begin{align*}
[P + Q] &= [P + Q] \\
[P + Q] &= \text{io}(([P] \cup [Q]) \cap A) \quad \text{Def. A4} \\
[P + Q] &= \text{io}(([P] \cup [Q]) \cap A) \quad \text{Def. A3} \\
[P + Q] &= \text{io}(([P] \cap A) \cup ([Q] \cap A)) \\
[P + Q] &= \text{io}(([P] \cap A) \cup ([Q] \cap A)) \text{ By (11)} \\
[P + Q] &= [P] \cup [Q] \quad \text{Def. A4}
\end{align*}
\]

Lemma B.2 If \( EL \) is an event library, and \( P \) and \( Q \) be basic state machines whose variables are disjoint from those in \( EL \), then \( [T'_{EL}]() \) is homomorphic w.r.t. concatenation, i.e.,

\[
[T'_{EL}([P]) \prec [T'_{EL}([Q])] = [T'_{EL}([P]) \prec [Q]]
\]

Proof: By (6)1 and (6)2, (6)3, and (6)4.

(5)7. Q.E.D.

Proof: By (5)5 and (5)6.

(4)3. Case: 4.1 \( \text{act}_{2,k} = (\epsilon, bx_\epsilon, oe_\epsilon, a'_\epsilon) \) for \( oe_\epsilon \in \text{OE} \), \( bx_\epsilon \in B\text{Exp} \cup \{\epsilon\} \), and \( a'_\epsilon \in \text{Assign} \cup \{\epsilon\} \)

Proof: Case assumption 4.1 contradicts assumption 2.8 since assumption 4.1 implies that \( io(a_2s_2) \neq io(a_1s_1) \) by definition of \( \langle \_ \rangle \) (Def. A.3) and definition of \( io \) (Eq. (11)). Hence, assumption 4.1 cannot hold.

(4)4. Q.E.D.

Proof: By (4)1, (4)2, (4)3 and definition of \( \text{Act} \) (Eq. (7)) and (Eq. (8)).

(3)3. Case: 3.1 \( \text{act}_{1,j} = (\epsilon, bx_\epsilon, oe, a_\epsilon) \) for \( oe \in \text{OE} \), \( bx_\epsilon \in B\text{Exp} \cup \{\epsilon\} \), and \( a_\epsilon \in \text{Assign} \cup \{\epsilon\} \)

Proof: Proof is the same as for (3)2.

(3)4. Q.E.D.

Proof: By (3)1, (3)2, (3)3 and definition of \( \text{Act} \) (Eq. (7)) and (Eq. (8)).

(2)2. Q.E.D.

Proof: By (2)1 and logical implication ( \( \Rightarrow \) ).

(1)3. Q.E.D.

Proof: By (1)2 and (1)2.
Proof of Lemma B.2

Assume: \( \var(EL) \cap (\var(Q) \cup \var(P)) = \emptyset \)

Prove: \( \llbracket T_{EL}'(P) \rrbracket = \llbracket T_{EL}'(Q) \rrbracket \)

(1.1) Choose \( Q' \in Q \) such that \( \llbracket Q \rrbracket = \llbracket Q' \rrbracket \), and \( \llbracket P \rrbracket = \llbracket P \rrbracket \)

Proof: Assume that all variables in \( Q \) are assigned to a value before they are used in a guard. Then execution of \( Q \) will be the same even if we choose an arbitrary initial data state. Therefore we must have that \( \llbracket P \rrbracket = \llbracket P \rrbracket \)

Therefore it may be the case that \( \llbracket P \rrbracket = \llbracket Q \rrbracket \) and \( \var(P) \cap \var(Q') \neq \emptyset \). In this case, executing \( P \) before \( Q \) will not affect the execution of \( Q' \). Therefore \( \llbracket P \rrbracket = \llbracket Q \rrbracket \).

(1.2) \( \llbracket T_{EL}'(P) \rrbracket = \llbracket T_{EL}'(Q) \rrbracket \)

Proof: By (1.1) and Lemma B.1

D.3 Composite transformations

Theorem B.2 The relation \( T_{EL}' \) is a function when restricted to the image of \( T_{EL} \), i.e.,

\[ \llbracket P \rrbracket = \llbracket Q \rrbracket \implies \llbracket T_{EL}'(P) \rrbracket \cap Im_{EL} = \llbracket T_{EL}'(Q) \rrbracket \cap Im_{EL} \]

Proof of Theorem B.2

Assume: \( \llbracket P_1, \ldots, P_n \rrbracket = \llbracket Q_1, \ldots, Q_n \rrbracket \)

Prove: \( \llbracket T_{EL}'(P_1, \ldots, P_n) \rrbracket \cap Im_{EL} = \llbracket T_{EL}'(Q_1, \ldots, Q_n) \rrbracket \cap Im_{EL} \)

(1.1) Assume: \( s' \in \llbracket T_{EL}'(P_1, \ldots, P_n) \rrbracket \cap Im_{EL} \)

Proof: By definition of \( rnm \) (see App. B.1).

(2.1) Choose \( EL_1 \in EL \) such that \( EL_1 = rnm(P_1, \ldots, P_n, EL) \)

Proof: By assumption of \( rnm \) (see App. B.1).

(2.2) Choose \( s' \in \llbracket T_{EL}'(P_1, \ldots, P_n) \rrbracket \) such that \( s' \in Im_{EL} \)

Proof: By assumption 1.1 and definition of \( [\cdot] \) (Def. A.5), and definition of \( T_{EL}' \) (Def. B.5).

(2.3) Choose \( s_1, \ldots, s_n \in \llbracket T_{EL}'(P_1, \ldots, P_n) \rrbracket \) such that \( s' \in Im_{EL_1} \)

Proof: By (2.2) (2.2) is a function) and Lemma B.1 (which ensures that \( T_{EL_1}'([\cdot]) \) is defined for singletons).

(2.4) Choose \( s \in \llbracket P_1, \ldots, P_n \rrbracket \) such that \( s = \llbracket s_1, \ldots, s_n \rrbracket \)

Proof: By (2.2) (2.3) definition of \( [\cdot] \) (Def. A.5) and definition of \( Im_{EL} \) (Eq. (16)).
(2)5. $s \in \| P_1 \| \cdots \| P_n \|

Proof: By (2)4 and assumption 1.

(2)6. $t_1 \in \| Q_1 \|, \ldots, t_n \in \| Q_n \|$ such that $s \in \| (t_1, \ldots, t_n) \|

Proof: By (2)3, (2)4, (2)5, definition of $\|$ (Def. A.5), and definition of composite state machine expressions Def. A.2.

(2)7. $t_i = s_i$ for all $i \in \{1, \ldots, n\}

Proof: By definition of a composite state machine, each basic state machine it consists of must have different names. Therefore $\| Q_i \| \cap \| Q_j \| \neq \emptyset$
for all $i, j \in \{1, \ldots, n\}$ such that $i \neq j$. By definition of parallel composition, there is one and only one trace $t_i$ in each basic state machine $Q_i$
(for $i \in \{1, \ldots, n\}$) such that $s \in \| (t_1, \ldots, t_n) \|$. The same holds for the basic state machines $P_i$, therefore we have that $s_i$ must be equal to $t_i$.

(2)8. Choose $EL_2 \in EL$ such that $EL_2 = \text{rnm}(Q_1 \| \cdots \| Q_n, EL)$

Proof: By definition of $\text{rnm}$ (see App. B.1).

(2)9. $s' \in \bigcup (T^{EL}_E((s_1)), \ldots, T^{EL}_E((s_n))) \cap \text{Im}_EL$

Proof: By (2)2, (2)3, (2)8, Theorem B.1 (which ensures that $\| T^{EL}_E(\_\|)$ is a function).

(2)10. Q.E.D.

Proof: By (2)2, (2)3, (2)6, (2)7, (2)9, and definition of $\|$ (Def. A.5).

Lemma B.3 The semantics of the event transformation for composite state machines induced by an event library $EL$ is homomorphic w.r.t. the union operator when restricted to its image, i.e.,

$$\hat{T}^{EL}(\| P \| \cup \| Q \|) = \hat{T}^{EL}(\| P \|) \cup \hat{T}^{EL}(\| Q \|)$$

when $\text{var}(EL) \cap \text{var}(P) \cap \text{var}(Q) = \emptyset$

Proof of Lemma B.3 Assume $\text{var}(EL) \cap \text{var}(P) \cap \text{var}(Q) = \emptyset$, then we have

$$\bigcup_{s \in \| P \| \cup \| Q \|} t^{C}_{EL}(s) = \bigcup_{s \in \| P \|} t^{C}_{EL}(s) \cup \bigcup_{s \in \| Q \|} t^{C}_{EL}(s)$$

$$\hat{T}^{EL}(\| P \| \cup \| Q \|) = \hat{T}^{EL}(\| P \|) \cup \hat{T}^{EL}(\| Q \|)$$

By Lemma B.3.1

Lemma B.3.1 Let the function $\hat{t}^{C}_{EL} \in E^* \rightarrow P(E^*)$ be defined by

$$\hat{t}^{C}_{EL}(s) \overset{\text{def}}{=} \| (T^{EL}_E((s|E_{n_1})), \ldots, T^{EL}_E((s|E_{n_m}))) \|

for all $s \in T$ and $\text{Nm} = \{nm_1, \ldots, nm_n\}$.

Then the semantics of the transformation $\hat{T}^{EL}(P)$ is entirely characterized by $\hat{t}^{C}_{EL}$ if $\text{var}(EL) \cap \text{var}(P) = \emptyset$, i.e.,

$$\hat{T}^{EL}(\| P \|) = \bigcup_{s \in \| P \|} t^{C}_{EL}(s)$$

Proof of Lemma B.3.1

Prove: $\hat{T}^{EL}(\| P \|) = \bigcup_{s \in \| P \|} t^{C}_{EL}(s)$

Let: $P \overset{\text{def}}{=} P_1 \| \cdots \| P_n$
D.4 Adherence preservation under event transformations

Theorem B.3 Let $r$ be a restriction, and $h$ be a high-level relation, and $\llbracket T'_{EL} \rrbracket$ be the transformation induced by event library $EL$. Then $T'_{EL}$ preserves $\text{Bsp}_{rb}$ for specification $P$ if the following conditions are satisfied for all $s, t, u \in \llbracket P \rrbracket$, $s' \in \hat{T}_{EL}(\{s\}), t' \in \hat{T}_{EL}(\{t\})$

$$s' \xrightarrow{r} t' \implies s \xrightarrow{r} t$$

$$s \xrightarrow{h} t \wedge s' \xrightarrow{h} u \sim_{l} t \implies \exists u' \in \hat{T}_{EL}(\{u\}) : s' \xrightarrow{h} u' \sim_{l} t'$$

Proof of Theorem B.3

Assume: 1. $\forall s, t \in \llbracket P \rrbracket, s' \in \hat{T}_{EL}(\{s\}), t' \in \hat{T}_{EL}(\{t\}) : s' \xrightarrow{r} t' \implies s \xrightarrow{r} t$

2. $\forall s, t, u \in \llbracket P \rrbracket, s' \in \hat{T}_{EL}(\{s\}), t' \in \hat{T}_{EL}(\{t\}) : s \xrightarrow{h} t \wedge s' \xrightarrow{h} u \sim_{l} t \implies \exists u' \in \hat{T}_{EL}(\{u\}) : s' \xrightarrow{h} u' \sim_{l} t$

Prove: $\text{Bsp}_{rb}(\llbracket P \rrbracket) \implies \text{Bsp}_{rb}(\hat{T}_{EL}(\llbracket P \rrbracket))$

(1) Assume: 1.1 $\text{Bsp}_{rb}(\llbracket P \rrbracket)$

1.2 $s' \xrightarrow{r} t'$ for $s', t' \in \hat{T}_{EL}(\llbracket P \rrbracket)$

Prove: $\exists u' \in \hat{T}_{EL}(\llbracket P \rrbracket) : s' \xrightarrow{h} u' \sim_{l} t'$
(2) 1. Choose \( s, t \in \llbracket P \rrbracket \) such that \( s \xrightarrow{c} t, s' \in \overline{T}_{EL}(\{s\}) \), and \( t' \in \overline{T}_{EL}(\{t\}) \).

Proof: By assumption 1, assumption 1.2, and Lemma. B.3.

(2) 2. Choose \( u \in \llbracket P \rrbracket \) such that \( s \xrightarrow{b} u \sim t \).

Proof: By assumption 1.1, (2)1 and definition of \( B_{\text{rh}} \) (Eq. (18)).

(2) 3. Choose \( u' \in \overline{T}_{EL}(\{u\}) \) such that \( s' \xrightarrow{b} u' \sim t' \).

Proof: By (2)1, (2)2, and assumption 2.

(2) 4. \( u' \in \overline{T}_{EL}(\llbracket P \rrbracket) \).

Proof: By (2)2, (2)3, and Lemma. B.3.

(2) 5. Q.E.D.

Proof: By (2)3 and (2)4.

(1) 2. Q.E.D.

Proof: By (1)1 and definition of \( B_{\text{rh}} \) (Eq. (18)).
Chapter 14

Paper 6: A Transformational Approach to Facilitate Monitoring of High Level Policies
A Transformational Approach to Facilitate Monitoring of High Level Policies

Fredrik Seehusen, Mass Soldal Lund, and Ketil Stølen

2009

Abstract

We present a method for specifying high level security policies that can be enforced by runtime monitoring mechanisms. The method has three main steps: (1) the user of our method formalizes a set of policy rules using UML sequence diagrams; (2) the user selects a set of transformation rules from a transformation library, and applies these using a tool to obtain a low level intermediate policy (also expressed in UML sequence diagrams); (3) the tool transforms the intermediate low level policy expressed in UML sequence diagrams into a UML inspired state machine that governs the behavior of a runtime policy enforcement mechanism. We believe that the method is both easy to use and useful since it automates much of the policy formalization process.

The method is underpinned by a formal foundation that precisely defines what it means that a system adheres to a policy expressed as a sequence diagram as well as a state machine. The foundation is furthermore used to show that the transformation from sequence diagrams to state machines is adherence preserving under a certain condition.

1 Introduction

Policies are rules governing the choices in the behavior of a system [19]. We consider the kind of policies that are enforceable by mechanisms that work by monitoring execution steps of some system which we call the target of the policy. This kind of mechanism is called an EM (Execution Monitoring) mechanism [16].

The security policy which is enforced by an EM mechanism is often specified by a state machine that describes exactly those sequences of security relevant actions that the target is allowed to execute. Such EM mechanisms receive an input whenever the target is about to execute a security relevant action. If the state machine of the EM mechanism has an enabled transition on a given input, the current state is updated according to where the transition lands. If the state machine has no enabled transition for a given input, then the target is about to violate the policy being enforced. It may therefore be terminated by the EM mechanism.

Security policies are often initially expressed as short natural language statements. Formalizing these statements is, however, time consuming since they often refer to high level notions such as “opening a connection” or “sending an SMS” which must ultimately be expressed as sequences of security relevant
actions of the target. If several policies refer to the same high level notions, or should be applied to different target platforms, then these must be reformalized for each new policy and each new target platform.

Clearly, it is desirable to have a method that automates as much of the formalization process as possible. In particular, the method should:

1. support the formalization of policies at a high level of abstraction;
2. offer automatic generation of low level policies from high level policies;
3. facilitate automatic enforcement by monitoring of low level policies;
4. be easy to understand and employ by the users of the method (which we assume are software developers).

The method we present has three main steps which accommodate the above requirements:

**Step I** The user of our method receives a set of policy rules written in natural language, and formalizes these using UML sequence diagrams.

**Step II** The user creates transformation rules (expressed in UML sequence diagrams) or selects them from a transformation library, and applies these using a tool to obtain an intermediate low level policy (also expressed in UML sequence diagrams).

**Step III** The tool transforms the intermediate low level policy expressed in UML sequence diagrams into state machines that govern the behavior of an EM mechanism.

There are two main advantages of using this method as opposed to formalizing low level policies directly using state machines. First, much of the formalization process is automated due to the transformation from high to low level. This makes the formalization process less time consuming. Second, it will be easier to show that the formalized high level policy corresponds to the natural language policy it is derived from, than to show this for the low level policy. The reason for this is that the low level policy is likely to contain implementation specific details which make the intention of the policy harder to understand.

The choice of UML is motivated by requirement 4. UML is widely used in the software industry. It should therefore be understandable to many software developers which are the intended users of our method. UML sequence diagrams are particularly suitable for policy specification in the sense that they specify partial behavior (as opposed to complete), i.e., the diagrams characterize example runs or snapshots of behavior in a period of time. This is useful when specifying policies since policies are partial statements that often do not talk about all aspects of the target’s behavior. In addition to this, UML sequence diagrams allow for the explicit specification of negative behavior, i.e., behavior that the target is not permitted to engage in. This is useful because the only kind of policies that can be enforced by EM mechanisms are prohibition policies, i.e., policies that stipulate what the target is not allowed to do.

The main body if this report gives an example driven presentation of our method without going into all the technical details. In the appendices, we present the formal foundation of the method. In particular, the rest of this
report is structured as follows: In Sect. 2 we describe step I of our method by introducing a running example and showing how high level security policies can be expressed with UML sequence diagrams. Sect. 3 describes step II of our method by showing how a transformation from high level to low level policies can be specified using UML sequence diagrams. Step III of our method is described in Sect. 4 which defines a transformation from (low level) sequence diagram policies to state machine policies that can be enforced by EM mechanisms. Sect. 4 discusses related work, and Sect. 5 provides conclusions and directions of future work. The formal foundation is presented in the appendices. App. A and App. B, present the syntax and the semantics of UML sequence diagrams and state machines, respectively. We also define adherence, i.e., what it means for a system to adhere to a sequence diagram or a state machine. Finally, App. C characterizes the transformation from high to low level sequence diagrams. App D defines the transformation from low level sequence diagrams to state machines. In App D, we also show that the transformation is adherence preserving given that a certain condition is satisfied. All proofs are given in App. E.

2 Step I: Specifying policies with sequence diagrams

In the first step of our method, the user receives a set of policy rules written in natural language. The user then formalizes these rules using UML sequence diagrams. In this section, we show how to express two security policies. The examples are taken from an industrial case study conducted in the EU project S3MS [17].

As the running example of this report, we consider applications on the Mobile Information Device Profile (MIDP) Java runtime environment for mobile devices. We assume that the runtime environment is associated with an EM mechanism that monitors the executions of applications. Each time an application makes an API-call to the runtime environment, the EM mechanism receives that method call as input. If the current state of the state machine that governs the EM mechanism has no enabled transitions on that input, then the application is terminated because it has violated the security policy of the EM mechanism.

2.1 Example – specifying policy 1

The first policy we consider is

*The application is not allowed to establish connections to other addresses than http://s3ms.fast.de.*

This policy is specified by the UML sequence diagram of Fig. 1.

Sequence diagrams describe communication between system entities which we will refer to as *lifelines*. In a diagram, lifelines are represented by vertical dashed lines. An arrow between two lifelines represents a message being sent from one lifeline to the other in the direction of the arrow. Sequence diagrams should be read from top to bottom; a message on a given lifeline should occur
before all messages that appear below it on the same lifeline (unless the messages are encapsulated by operators). Communication is asynchronous, thus we distinguish between the occurrence of a message transmission and a message reception. Both kinds of occurrences are viewed as instantaneous and in the following called events.

The two lifelines in Fig. 1 are Application representing the target of the policy, and url, representing an arbitrary address that the target can connect to. The sending of message connect from the Application to url represents an attempt to open a connection.

Expressions of the form \{bx\} (where bx is a boolean expression) are called constraints. Intuitively, the interaction occurring below the constraint will only take place if the constraint evaluates to true.

The constraint of Fig. 1 should evaluate to true if and only if url is not equal to the address “http://s3ms.fast.de” (which according to the policy is the only address that the application is allowed to establish a connection to).

Interactions that are encapsulated by the neg operator specify negative behavior, i.e., behavior which the target is not allowed to engage in. Thus Fig. 1 should be read: Application is not allowed to connect to the arbitrary address url if url is different from the address “http://s3ms.fast.de”.

UML sequence diagrams are partial in the sense that they typically don’t tell the complete story. There are normally other legal and possible behaviors that are not considered within the described interaction. In particular, sequence diagrams explicitly describe two kinds of behavior: behavior which is positive in the sense that it is legal, valid, or desirable, and behavior which is negative meaning that it is illegal, invalid, or undesirable. The behavior which is not explicitly described by the diagram is called inconclusive meaning that it is considered irrelevant for the interaction in question.

We interpret sequence diagrams in terms of positive and negative traces, i.e., sequences of events (see App. A). When using sequence diagrams to express prohibition policies, we are mainly interested in traces that describe negative behavior. If a system is interpreted as a set of traces, then we say that the systems adheres to a policy if none of the system’s traces have a negative trace of a lifeline of the policy as a sub-trace\(^1\). Thus we take the position that the target is allowed to engage in (inconclusive) behavior which is not explicitly described by a given policy. This is reasonable since we do not want to use

\(^1\)A trace s is a (possibly non-continuous) sub-trace of trace t if s can be obtained from t by removing zero or more events from t.
policies to express the complete behavior of the target.

Turning back to the example, an application is said to adhere to the policy of Fig. 1 if none of its traces contain the transmission of the message connect to an address which is different from http://s3ms.fast.de.

### 2.2 Example – specifying policy 2

The second natural language policy is

*The application is not allowed to send more than N SMS messages (where N is a natural number).*

This policy is specified by the sequence diagram of Fig. 2. Again, the life-lines of the diagram are Application representing the policy target, and url, this time representing an arbitrary recipient address of an SMS message.

Boxes with rounded edges contain assignments of variables to values. In Fig. 2, the variable s is initialized to zero, and incremented by one each time the application sends an SMS. The loop operator is used to express the iteration of the interaction of its operand.

The alt-operator is used to express alternative interaction scenarios. In Fig. 2, there are two alternatives. The first alternative is applicable if the variable s is less than or equal to N (representing an arbitrary number). In this case, the application is allowed to send an SMS, and the variable s is incremented by one. The second alternative is applicable when s is greater than N. In this case, the application is not allowed to send an SMS as specified by the neg-operator.

An application adheres to the policy of Fig. 2 if none of its traces contain more than N occurrences of the message sendSMS.
3 Step II: Specifying transformations with sequence diagrams

In the second step of our method, the user creates new transformation rules or selects them from a transformation library. The users then employs a tool which automatically applies the transformation rules to the high level policy such that an intermediate low level policy is produced. In the following, we show how a transformation to the low level can be defined using UML sequence diagrams. An advantage of using sequence diagrams for this purpose is that the writer of the transformation rules can express the low level policy behavior using the same language that is used for writing high level policies. This will also make it easier for the user of our method to understand or modify the transformation rules in the transformation library if that should become necessary.

A transformation rule is specified by a pair of two diagram patterns (diagrams that may contain meta variables), one left hand side pattern, and one right hand side pattern. When a transformation rule is applied to a diagram \( d \), all fragments of \( d \) that match the left hand side pattern of the rule are replaced by the right hand side pattern. Meta variables are bound according to the matching. A diagram pattern \( dp \) matches a diagram \( d \) if the meta variables of \( dp \) can be replaced such that the resulting diagram is syntactically equivalent to \( d \).

In the following we illustrate the use of transformation rules by continuing the example of the previous section.

3.1 Example – specifying a transformation for policy 1

The policies described in the previous section are not enforceable since the behavior of the target is not expressed in terms of API-calls that can be made to the MIDP runtime environment. Recall the policy of Fig. 1. It has a single message connect which represents an attempt to open a connection. In order to make the policy enforceable, we need to express this behavior in terms of API-calls that can be made to the runtime environment.

Fig. 3 illustrates a transformation rule which describes how the connect message is transformed into the API-calls which can be made in order to establish a connection via the MIDP runtime environment. The diagram on the left in Fig. 3 represents the left hand side pattern of the rule, and the diagram on the right represents the right hand side pattern of the rule. In the diagram, all meta variables are underlined.

Fig. 4 shows the result of applying the rule of Fig. 3 to the policy of Fig. 1. Here we see that the message connect has been replaced by the relevant API-calls of the runtime environment.

Clearly, the high level policy of Fig. 1 allows for an easier comparison to the natural language description than the low level policy of Fig. 4. Moreover, if the high level policy of Fig. 1 should be applied to a different runtime environment than MIDP, say .NET, then a similar transformation rule can be written or selected from a transformation library without changing the high level policy.

Notice that each message of Fig. 4 contains one or more variables that are not explicitly assigned to any value in the diagram. We call these parameter variables, and distinguish these from normal variables by writing them in bold-
face. Parameter variables are bound to an arbitrary value upon the occurrence of the message they are contained in. Parameter variables differ from normal variables that have not been assigned to a value in that parameter variables contained in a loop are bound to new arbitrary values for each iteration of the loop.

The introduction of parameter variables constitutes an extension of the UML 2.1 sequence diagram standard. This extension is necessary in order to express policies for execution mechanisms that monitor method calls where the actual arguments of the method call are not known until the method call is intercepted. Without parameter variables, one would in many cases be forced to specify all possible actual arguments that a method call can have. This is clearly not feasible.

4 Step III: Transforming sequence diagrams to state machines

In the third step of our method, the low level intermediate sequence diagram obtained from step II is automatically transformed into a set of state machines (one for each lifeline of the diagram) that govern the behavior of an EM mechanism. The state machines explicitly describe the (positive) behavior which is allowed by a system. Everything which is not described by the state machines is (negative) behavior which is not allowed. Therefore, the state machines do not have a notion of inconclusive behavior as sequence diagrams do.

The semantics of a state machine is a set of traces describing positive behavior. A system adheres to a set of state machines $S$ if each trace described by the system is also described by a state machine in $S$ (when the trace is restricted to the alphabet of the state machine). We define adherence like this because this
The transformation of step III should convert a sequence diagram into a set of state machine such that an arbitrary system adheres to the state machine set if and only if the system adheres to the sequence diagram, i.e., the transformation should be adherence preserving. As one would except, the transformation converts positive and negative behavior of the sequence diagram into positive and negative behavior of the state machines, respectively. However, the inconclusive behavior of the sequence diagram is converted to positive behavior of the state machines. This is because, by definition of adherence for sequence diagrams, a system is allowed to engage in the (inconclusive) behavior which is not described by the sequence diagram.

The only kind of policies that can be enforced by EM mechanisms are so-called prohibition policies, i.e., policies that describe what a system is not allowed to do. Sequence diagrams are suitable for specifying these kinds of policies because they have a construct for specifying explicit negative behavior. For this reason, the sequence diagram policies are often more readable than the corresponding state machine policies since the state machine policies can only explicitly describe behavior which is allowed by an application. In the following examples, we clarify this point and explain how the transformation from sequence diagrams to state machines works.

### 4.1 Example – transforming policy 2 to a state machine

Although our method is intended to transform low level intermediate sequence diagram policies to state machines, we will use the sequence diagram of Fig. 2 to illustrate the transformation process as this diagram better highlights the transformation phases than the low level intermediate policy of Fig. 4.
STEP III: TRANSFORMING SEQUENCE DIAGRAMS TO STATE MACHINES

In general, the transformation from a sequence diagram yields one state machine for each lifeline of the diagram. However, in the current example, the target of the policy is the lifeline Application. Thus we only consider the transformation of this lifeline.

The transformation from a sequence diagram \( d \) with one lifeline to a state machine has two phases. In phase 1, the sequence diagram \( d \) is transformed into a state machine \( SM \) whose trace semantics equals the negative trace set of \( d \). In phase 2, \( SM \) is inverted into the state machine \( SM' \) whose semantics is the set of all traces that do not have a trace of \( SM \) as a sub-trace.

In the following we explain the two phases by transforming a diagram describing lifeline Application in Fig. 2 into a state machine.

4.1.1 Phase 1

First, in phase 1, we transform the sequence diagram describing the lifeline Application into a state machine whose trace semantics equals the negative traces of the diagram. To achieve this, we make use of the operational semantics of sequence diagrams which is based on [13, 12] and described in App. A.

The operational semantics makes use of a so-called projection system for finding enabled events and constructs in a diagram. The projection system is a labeled transition system (LTS) whose states are diagrams, and whose transitions are labeled by events, constraints, assignments, and so-called silent events that indicate which kind of operation has been executed. If the labeled transition system has a transition from a diagram \( d \) to a diagram \( d' \) that is labeled by, say, event \( e \), then we understand that event \( e \) is enabled in diagram \( d \), and \( d' \) is obtained by removing \( e \) from \( d \).

To transform a diagram describing Application into a state machine describing its negative traces, we first construct the projection system whose states are exactly those that can be reached from the diagram. The result is illustrated in Fig. 5 (note that the labels \( \tau_{alt} \), \( \tau_{loop} \), and \( \tau_{neg} \) are silent events that correspond to the sequence diagram constructs \( alt \), \( loop \), and \( neg \), respectively). Then, the projection system is transformed into a state machine describing the negative traces of the diagram. The states of this state machine are of the form \( (Q, mo) \) where \( Q \) is a set of states of the projection system and \( mo \) is a mode which is used to differentiate between positive and negative traces. There are two kinds of modes: \( pos \) (for positive) or \( neg \) (for negative). A state with mode \( neg \) leads to a final state that accepts negative traces. Thus we require that all final states have mode \( neg \).

When converting the projection system into a state machine, we remove all silent events and concatenate constraints with succeeding events and assign-
ments with preceding events. In addition, the silent event $\tau_{neg}$ is used to find negative executions. That is, any execution that involves a $\tau_{neg}$ represents a negative execution that lead to states having mode $neg$.

The result of converting the projection system of Fig. 5 into a basic state machine is illustrated in Fig. 6. Here the black filled circle represents an initial state, the boxes with rounded edges represent simple states, and the black filled circle encapsulated by another circle represents a final state. Transitions of state machines are labeled by action expressions of the form $nm.si[bx]/ef$. Here $nm.si$ (where $si$ denotes a signal and $nm$ denotes the name of the state machine the signal is received from) is called an input event, $[bx]$ (where $bx$ is a boolean expression) is called a guard, and $ef$ is called an effect. An effect is a sequence of assignments and/or an output event.

The graphical notation used for specifying state machines is inspired by the UML statechart diagram notion. See App. B for more details.

The alphabet of a state machine is a set of events. When we transform a sequence diagram to a state machine, the alphabet of the state machine is the set of all events that occur in the sequence diagram.

In App.D.1.1, we formalize the transformation of phase 1. We also prove that this transformation is correct in the sense the state machine describes exactly the negative traces of the sequence diagram it is transformed from.

### 4.1.2 Phase 2

In phase 2, we “invert” the state machine of Fig. 6 into the state machine whose semantics is the set of all traces that do not have a trace of the state machine of Fig. 6 as a sub-trace. In general, the inversion $SM'$ of a state machine $SM$ has the power-set of the states in $SM$ as its states\(^2\) the alphabet of $SM$ as its alphabet, all its states as its final states, and its transitions are those that are constructed by the following rule

- if $Q$ is a state of $SM'$, then $SM'$ has a transition $Q \overset{act}{\rightarrow} Q \cup Q'$ where $Q'$ represents the set of non-final states of $SM$ that are targeted by an outgoing transition of a state in $Q$ whose action expression contains the same event as $act$.

\(^2\)Each state of $SM'$ also needs to keep track of the transitions already visited in $SM$ to reach that state. We postpone the discussion of this technical detail to App. D
4 STEP III: TRANSFORMING SEQUENCE DIAGRAMS TO STATE MACHINES

Figure 7: State machine describing positive and inconclusive traces for Application

Figure 8: Composition of three sequence diagram policies

Note that the rule is slightly simplified as a lengthy presentation of all the technical details of the rule is given in App. D.1.2. In the appendix, we also show that the transformation of phase 2 correctly inverts policies that may be seen as compositions of (sub)policies with disjoint sets of variables. The state machine policy of Fig. 6, can not be seen as a composition of more than one policy. Therefore the condition of disjoint variables is trivially satisfied for this state machine.

Returning back to our running example, Fig. 7 shows the inversion of the state machine of Fig. 6. Since all states in the state machine of Fig. 7 are final, we have omitted to specify the final states.

We have that every trace that contain less than or equal to $N$ occurrences of the sendSMS message are accepted by the state machine of Fig. 7. Indeed, this was the intended meaning of the policy.

In App.D.2, we show that the composition of the transformations of phase 1 and phase 2 is adherence preserving when the condition under which the transformation of phase 2 yields a correct inversion is satisfied.

4.2 Example – why the negation construct is useful

As noted in the beginning of this section, the sequence diagram construct for specifying explicit negative behavior is useful when specifying policies that can be enforced by EM mechanisms. In this section, we illustrate this with an
Consider the policy shown in Fig. 8. It may be seen as a composition of three policies. The upper most policy states that after the lifeline $A$ has transmitted $a$, it is not allowed to transmit $b$. The two other policies are similar except that the messages are different.

To transform a diagram describing the lifeline $A$ into a state machine, we first (in phase 1) construct the state machine that describes the negative traces of the diagram. The resulting state machine is shown in Fig. 9. Then, we invert the state machine of Fig. 9 to obtain the state machine of Fig. 10.

Clearly, it is more difficult to understand the meaning of the state machine policy of Fig. 10 than the sequence diagram policy of Fig. 8. The reason for this is that the state machine policy have to describe all behavior which is allowed. However, the process of inverting a state machine may lead an increase in the number of states and transitions. This shows why it is useful to have a construct for specifying negative behavior.

5 Related work

Previous work that address the transformation of policies or security requirements are [1, 2, 3, 4, 6, 7, 11, 14, 15, 21]. All these differ clearly from ours in that the policy specifications, transformations, and enforcement mechanisms are different from the ones considered in this report.
Bai and Varadharajan [1] consider authorization policies, Satoh and Yamaguchi [15] consider security policies for Web Services, Patz et. al. [14] consider policies in the form of logical conditions, and Beigi et.al. [4] focuses on transformation techniques rather than any particular kind policies. The remaining citations ([2, 3, 6, 7, 11, 21]) all address policies in the form of access-control requirements.

Of the citations above, [3] gives the most comprehensive account of policy transformation. In particular it shows how platform independent role based access control requirements can be expressed in UML diagrams, and how these requirements can be transformed to platform specific access control requirements.

The transformation of sequence diagrams (or a similar language) to state machines has been previously addressed in [5, 9, 22]. However, these do not consider policies, nor do they offer a way of changing the granularity of interactions during transformation.

The only paper that we are aware of that considers UML sequence diagrams for policy specification is [20]. However, in that paper, transformations from high- to low level policies or transformation to state machines is not considered. The paper argues that sequence diagrams must be extended with customized expressions for deontic modalities to support policy specification. While this is true in general, this is not needed for the kind of prohibition policies that can be enforced by EM mechanisms.

6 Conclusions

We claim that it is desirable to automate as much as possible of the process of formalizing security policies. To this end we have presented a method which (1) supports the formalization of policies at a high level of abstraction, (2) offers automatic generation of low level policies from high level policies, and (3) facilitates automatic enforcement by monitoring of low level policies. Enforcement mechanisms for the kind of policies considered in this report have been developed in the S3MS EU project [17]. Thus the method fulfills the first three requirements that were presented in the Sect. 1. Empirical investigation of whether the method satisfies the fourth requirement, namely that it should be easy to understand by software developers, is beyond the scope of this report. However, we have used UML as a policy language, and using UML for specifying policies, we believe, is not much harder than using UML to specify software systems (in particular, since we focus on execution monitoring and do not have to take other modalities than prohibition into consideration).

In the appendices, we provide a formal foundation for our method. In particular, we define the semantics of sequence diagrams and state machines, and we precisely define what it means that a system adheres to a sequence diagram policy as well as a state machine policy. We also formalize the transformation from high to low level sequence diagrams, and the transformation from sequence diagrams to state machines. Finally, we prove that the transformation from sequence diagram policies to state machine policies is adherence preserving under a certain condition. All examples of this report satisfies this condition.

Previous work in the literature has addressed policy transformation, but differ clearly from ours in that the policy specifications, transformations, and
enforcement mechanisms are different.

Acknowledgement

The research of this report has partly been funded by the Research Council of Norway project SECURIS (152839/220) and DIGIT (180052/S10), and partly by the EU through the S3MS (contract no. 27004) project.

References


REFERENCES


A  UML sequence diagrams

In this section, we first present the syntax (Sect. A.1) and semantics (Sect. A.2) of UML sequence diagrams. Then, in Sect. A.3, we define what it means for a system to adhere to a sequence diagram policy.

A.1 Syntax

We use the following syntactic categories to define the textual representation of sequence diagrams:

\[
\begin{align*}
ax & \in \text{AExp} \quad \text{arithmetic expressions} \\
bx & \in \text{BExp} \quad \text{boolean expressions} \\
sx & \in \text{SExp} \quad \text{string expressions}
\end{align*}
\]

We let \( \text{Exp} \) denote the set of all arithmetic, boolean, and string expressions, and we let \( ex \) range over this set. We denote the empty expression by \( \epsilon \). We let \( \text{Val} \) denote the set of all values, i.e., numerals, strings, and booleans (\( t \) or \( f \)) and we let \( \text{Var} \) denote the set of all variables. Obviously, we have that \( \text{Val} \subset \text{Exp} \) and \( \text{Var} \subset \text{Exp} \).

Every sequence diagram is built by composing atoms or sub-diagrams. The atoms of a sequence diagram are the events, constraints, and the assignments. These constructs are presented in Sect. A.1.1, while the syntax of sequence diagrams in general is presented in Sect. A.1.2. Finally, in Sect. A.1.3, we present syntax constraints for sequence diagrams.

A.1.1 Events, constraints, and assignments

The atoms of a sequence diagram are the events, constraints, and the assignments. An event is a pair \((k, m)\) of a kind \(k\) and a message \(m\). An event of the form \((!, m)\) represents a transmission of message \(m\), whereas an event of the form \((?, m)\) represents a reception of \(m\). We let \( \text{E} \) denote the set of all events:

\[
\text{E} \overset{\text{def}}{=} \{!, ?, \} \times \text{M}
\]

where \( \text{M} \) denotes the set of all messages.

On events, we define a kind function \( k_\cdot \in \text{E} \to \{!, ?\} \) and a message function \( m_\cdot \in \text{E} \to \text{M} \):

\[
k.(k, m) \overset{\text{def}}{=} k \\
m.(k, m) \overset{\text{def}}{=} m
\]

Messages are of the form \((l_t, l_r, si)\) where \(l_t\) represents the transmitter lifeline of the message, \(l_r\) represents the receiver lifeline of the message, and \(si\) represents the signal of the message. We let \( \text{L} \) denote the set of all lifelines, and \( \text{SI} \) denote the set of all signals. The set \( \text{M} \) of all messages is then defined by

\[
\text{M} \overset{\text{def}}{=} \text{L} \times \text{L} \times \text{SI}
\]

On messages, we define a transmitter function \( tr_\cdot \in \text{M} \to \text{L} \) and a receiver function \( re_\cdot \in \text{M} \to \text{L} \):

\[
tr.(l_t, l_r, si) \overset{\text{def}}{=} l_t \\
re.(l_t, l_r, si) \overset{\text{def}}{=} l_r
\]

(4)
We let the transmitter and receiver functions also range over events, \( tr, re \in E \rightarrow L \):

\[
tr(k, m) \overset{\text{def}}{=} tr.m \\
re(k, m) \overset{\text{def}}{=} re.m
\]

We define a lifeline function \( l \in E \rightarrow L \) that returns the lifeline of an event and a function \( l^{-1} \in E \rightarrow L \) that returns the inverse lifeline of an event (i.e., the receiver of its message if its kind is transmit and the transmitter of its message if its kind is receive):

\[
l.e \overset{\text{def}}{=} \begin{cases} tr.e & \text{if } k.e =! \\ re.e & \text{if } k.e =? \end{cases} \\
l^{-1}.e \overset{\text{def}}{=} \begin{cases} tr.e & \text{if } k.e =? \\ re.e & \text{if } k.e =! \end{cases}
\]

A signal is a tuple \((nm, ex_1, \ldots, ex_n)\) where \( nm \) denotes the signal name, and \( ex_1, \ldots, ex_n \) are the parameters of the signal. We usually write \( nm(ex_1, \ldots, ex_n) \) instead of \((nm, ex_1, \ldots, ex_n)\). Formally, the set of all signals is defined

\[
SI \overset{\text{def}}{=} Nm \times \text{Exp}^*\]

where \( A^* \) yields the set of all sequences over the elements in the set \( A \).

A signal may contain special so-called parameter variables that are bound to values upon the occurrence of the signal. Parameter variables are similar to free normal variables (normal variables that have not explicitly been assigned to a value). However, they differ in that parameter variables contained in a loop will be assigned to new values for each iteration of the loop.

A parameter variable is a pair \((vn, i)\) consisting of variable name \( vn \) and an index \( i \) (this is a natural number). When a sequence diagram is executed, the index of a parameter variable contained in a loop will be incremented by one for each iteration of the loop. This is to ensure that the parameter variable is given a new value when the loop is iterated. Hence, the index of a parameter variable is only used for bookkeeping purposes during execution, and it will never be explicitly specified in a graphical diagram.

In a graphical sequence diagram, parameter variables are distinguished from normal variables by writing the parameter variables in boldface. The index of a parameter variable in a graphical sequence diagram is always initially assumed to be zero.

The set of all parameter variables \( P\text{Var} \) is defined

\[
P\text{Var} \overset{\text{def}}{=} VN \times \mathbb{N}\]

where \( VN \) is the set of all variable names and \( \mathbb{N} \) is the set of all natural numbers. We assume that

\[
P\text{Var} \subset \text{Var}\]

A constraint is an expression of the form

\[
\text{constr}(bx, l)
\]

where \( bx \) is a boolean expression and \( l \) is a lifeline. Intuitively, interactions occurring after a constraint in a diagram will only take place if and only if the boolean expression of the constraint evaluates to true. We denote the set of all constraints by \( C \) and we let \( c \) range over this set.

An assignment is an expression of the form

\[
\text{assign}(x, ex, l)
\]
where $x$ is a normal variable, i.e., $x \in \text{Var} \setminus \text{PVar}$, $ex$ is an expression, and $l$ is a lifeline. Intuitively, the assignment represents the binding of expression $ex$ to variable $x$ on lifeline $l$. We let $A$ denote the set of all assignments and we let $a$ range over this set.

We define the function $l._{\text{const}}(bx, l) = l$ and $l._{\text{assign}}(x, ex, l) = l$ (10)

We denote by $E^l$, $C^l$, and $A^l$, the set of all events, constraints, and assignments with lifeline $l$, respectively, i.e.,

$$E^l = \{e \mid l.e = l\} \quad C^l = \{c \mid l.c = l\} \quad A^l = \{a \mid l.a = l\}$$

A.1.2 Diagrams

In the previous section, we presented the atomic constructs of a sequence diagram. In this section, we present the syntax of sequence diagrams in general.

**Definition 1 (Sequence diagram)** Let $e$, $bx$, $l$, $x$, and $ex$ denote events, boolean expressions, lifelines, variables, and expressions, respectively. The set of all syntactically correct sequence diagram expressions $D$ is defined by the following grammar:

$$d ::= \text{skip} \mid e \mid \text{constr}(bx, l) \mid \text{assign}(x, ex, l) \mid \text{refuse}(d) \mid \text{loop}(0..*) (d) \mid d_1 \text{ seq } d_2 \mid d_1 \text{ alt } d_2 \mid d_1 \text{ par } d_2$$

The base cases implies that any event ($e$), skip, constraint ($\text{constr}(bx, l)$), or assignment ($\text{assign}(x, ex, l)$) is a sequence diagram. Any other sequence diagram is constructed from the basic ones through the application of operators for negation ($\text{refuse}(d)$), iteration ($\text{loop}(0..*) (d)$), weak sequencing ($d_1 \text{ seq } d_2$), choice ($d_1 \text{ alt } d_2$), and parallel execution ($d_1 \text{ par } d_2$).

We define some functions over the syntax of diagrams. We let the function $eca.\alpha : D \rightarrow \mathcal{P}(E \cup C \cup A)$ return all events, constraints, and assignments present in a diagram. The function is defined as follows

$$eca.\alpha = \{\alpha\} \quad \text{for } \alpha \in E \cup C \cup A$$
$$eca.\text{skip} = \emptyset$$
$$eca.(op(d)) = eca.d \quad \text{for } op \in \{\text{refuse, loop}(0..*)\}$$
$$eca.(d_1 \text{ op } d_2) = eca.d_1 \cup eca.d_2 \quad \text{for } op \in \{\text{seq, alt, par}\}$$

Note that we henceforth let $\alpha$ denote an arbitrary event, constraint, or assignment, i.e., $\alpha \in E \cup C \cup A$.

The function $ll.d : D \rightarrow \mathcal{P}(L)$ returns all lifelines of a diagram:

$$ll.d = \bigcup_{\alpha \in eca.d} \{l.\alpha\}$$

We denote by $D^l$, the set of all diagrams with only one lifeline $l$, i.e.,

$$D^l = \{d \in D \mid ll.d = \{l\}\}$$
The function \( \text{msg} \in \mathcal{D} \to \mathcal{P}(\mathcal{M}) \) returns all the messages of a diagram:

\[
\text{msg.d} \equiv \bigcup_{e \in (\text{eca.d} \cap \mathcal{E})} \{m.e\}
\]  

(15)

The projection operator \( \pi \in \mathcal{L} \times \mathcal{D} \to \mathcal{D} \) that projects a diagram to a lifeline is defined

\[
\begin{align*}
\pi_l(\alpha) & \equiv \alpha & \text{if } l.\alpha = l \\
\pi_l(\alpha) & \equiv \text{skip} & \text{if } l.\alpha \neq l \\
\pi_l(\text{skip}) & \equiv \text{skip} \\
\pi_l(\text{op } d) & \equiv \text{op}(\pi_l(d)) & \text{for } \text{op} \in \{\text{refuse, loop}\langle0..*\rangle\} \\
\pi_l(d_1 \text{ op } d_2) & \equiv \pi_l(d_1) \text{ op } \pi_l(d_2) & \text{for } \text{op} \in \{\text{seq, alt, par}\}
\end{align*}
\]  

(16)

We let \( \text{var} \in (\text{Exp} \cup \mathcal{M}) \to \mathcal{P}(\mathcal{V ar}) \) be the function that yields the variables in an expression or the variables in the arguments of a signal of a message. We lift the function to diagrams as follows

\[
\begin{align*}
\text{var}(d) & \equiv \bigcup_{m \in \text{msg.d}} \text{var}(m) \cup \bigcup_{\text{constr}(bx,l) \in \text{eca.d} \cap \mathcal{C}} \text{var}(bx) \cup \\
& \quad \bigcup_{\text{assign}(x,ex,l) \in \text{eca.d} \cap \mathcal{A}} (\{x\} \cup \text{var}(ex))
\end{align*}
\]  

(17)

A.1.3 Syntax constraints

We impose some restrictions on the set of syntactically correct sequence diagrams \( \mathcal{D} \). We describe four rules which are taken from [12]. First, we assert that a given event should syntactically occur only once in a diagram. Second, if both transmitter and the receiver lifelines of a message are present in a diagram, then both the transmit event and the receive event of that message must be in the diagram. Third, if both the transmit event and the receive event of a message are present in a diagram, then they have to be inside the same argument of the same high level operator. The constraint means that in the graphical notion, messages are not allowed to cross the frame of a high level operator or the dividing line between the arguments of a high level operator. Fourth, the operator \( \text{refuse} \) is not allowed to be empty, i.e., to contain only the \( \text{skip} \) diagram.

The four rules described above are formally defined in [12]. These rules ensure that the operational semantics is sound and complete with the denotational semantics of sequence diagrams as defined in [12]. In this report, we define nine additional rules and we say that a diagram \( d \) is well formed if it satisfies these:

SD1 The variables of the lifelines of \( d \) are disjoint.

SD2 All parameter variables of \( d \) have index 0.

SD3 If \( m \) is a message in \( d \), then the arguments of the signal of \( m \) must be distinct parameter variables only.

SD4 The first atomic construct of each lifeline in \( d \) must be an assignment (not a constraint or an event).

SD5 All parameter variables that occur inside a loop in \( d \) do not occur outside that loop.

SD6 All loops in \( d \) must contain at least one event.
SD7 No two events in $d$ contain the same parameter variables.

SD8 For each lifeline in $d$, each constraint $c$ must be followed by an event $e$ (not an assignment or a constraint). In addition, the parameter variables of $c$ must be a subset of the parameter variables of $e$.

SD9 For each lifeline in $d$, the parameter variables of an assignment must be a subset of parameter variables of each event that proceeds it on the lifeline. If the assignment has no proceeding events on the lifeline, then the assignment cannot contain parameter variables.

SD10 All variables in $d$ (except for the parameter variables) must explicitly be assigned to a value before they are used.

The purpose of the syntax constraints is to ensure that the sequence diagram can be correctly transformed into a state machine.

Note that any graphical sequence diagram can be described by a textual diagram that satisfies conditions SD1 - SD4.

To obtain a diagram that satisfies SD1 and SD2 we have to rename variables on each lifeline and set the index of all parameter variables to zero. To obtain a diagram that satisfies condition SD3 we convert arguments (that are not parameter variables) of the signal of an event into constraints proceeding the event. For instance, the diagram

$$(!, (l_t, l_r, msg(ex)))$$

– which does not satisfy SD3 because $ex$ might not be a parameter variable – can be converted into the diagram

$$\text{constr}(px=ex, l_t) \text{ seq } (!, (l_t, l_r, msg(px))) \text{ for some } px \in \text{PVar}$$

which does satisfy SD3. Here $px=ex$ is a boolean expression that yields true if and only if $px$ is equal to $ex$.

If a sequence diagram $d$ does not satisfy condition SD4, then a dummy assignment can be added to start of each lifeline in $d$ that assigns some value to a variable that is not used in $d$.

A.2 Semantics

In this section, we define the operational semantics of UML sequence diagrams based on the semantics defined in [12]. The operational semantics tells us how a sequence diagram is executed step by step. It is defined as the combination of two labeled transition systems, called the execution system and the projection system.

These two systems work together in such a way that for each step in the execution, the projection system updates the execution system by selecting an enabled event to execute and returning the state of the diagram after the execution of the event.

A.2.1 The projection system

The projection system is used for finding enabled events at each step of execution. The projection system (as well as the execution system) is formally described by a labeled transition system (LTS).
Definition 2 (Labeled transition system (LTS)) A labeled transition system over the set of labels $LE$ is a pair $(Q, R)$ consisting of

- a (possibly infinite) set $Q$ of states;
- a ternary relation of $R \subseteq (Q \times LE \times Q)$, known as a transition relation.

We usually write $q \xrightarrow{le} q' \in (Q, R)$ if $(q, le, q') \in R$, or just $q \xrightarrow{le} q'$ if $(Q, R)$ is clear from the context. If $s = \langle le_1, le_2, \ldots, le_n \rangle$, we write $q \xrightarrow{s} q'$ for $q \xrightarrow{le_1} q_1 \xrightarrow{le_2} q_2 \cdot \cdot \cdot \xrightarrow{le_n} q'$. For the empty sequence $\langle \rangle$, we write $q \xrightarrow{\langle \rangle} q'$ iff $q = q'$.

To define the projection system, we make use of a notion of structural congruence which defines simple rules under which sequence diagrams should be regarded as equivalent.

Definition 3 (Structural congruence) Structural congruence over sequence diagrams, written $\equiv$, is the congruence over $D$ determined by the following equations:

1. $d \text{ seq skip } \equiv d$, $\text{skip seq } d \equiv d$
2. $d \text{ par skip } \equiv d$, $\text{skip par } d \equiv d$
3. $\text{skip alt } \equiv \text{skip}$
4. $\text{loop}(0..*) (\text{skip}) \equiv \text{skip}$

The projection system is an LTS whose states are pairs $\Pi(L, d)$ consisting of a set of lifelines $L$ and a diagram $d$. If the projection system has a transition from $\Pi(L, d)$ to $\Pi(L, d')$ that is labeled by, say event $e$, then we understand that $e$ is enabled in diagram $d$, and that $d'$ is obtained from $d$ by removing event $e$. Whenever the high level construct $\text{alt}$, $\text{refuse}$, or $\text{loop}$ is enabled in a diagram, the projection system will produce a so-called silent event that indicates the kind of construct that has been executed. For instance, each state of the form $\Pi(L, \text{refuse}(d))$ has a transition to $\Pi(L, d)$ that is labeled by the silent event $\tau_{\text{refuse}}$.

The set of lifelines $L$ that appears in the states of the projection system is used to define the transition rules of the weak sequencing operator $\text{seq}$. The weak sequencing operator defines a partial order on the events in a diagram; events are ordered on each lifeline and ordered by causality, but all other ordering of events is arbitrary. Because of this, there may be enabled events in both the left and the right argument of a $\text{seq}$ if there are lifelines present in the right argument of the operator that are not present in the left argument. The set of lifelines $L$ is used to keep track of which lifelines are shared by the arguments of $\text{seq}$, and which lifelines only occur in the right argument (but not the left) of $\text{seq}$.

The following definition of the projection system is based on [12].

Definition 4 (Projection system) The projection system is an LTS over $\alpha_t \in \{\tau_{\text{refuse}}, \tau_{\text{alt}}, \tau_{\text{loop}}\} \cup E \cup C \cup A$ whose states are $\Pi(\_ \_ \_ ) \in P(L) \times D$ and whose transitions are exactly those that can be derived by the following rules
A UML SEQUENCE DIAGRAMS

The revised projection system that handles parameter variables is the LTS over

$$\Pi(L, \alpha) \xrightarrow{\tau_{alt}} \Pi(L, \text{skip})$$ if $l.\alpha \in L$

$$\Pi(L, \text{refuse} (d)) \xrightarrow{\tau_{refuse}} \Pi(L, d)$$ if $ll.d \cap L \neq \emptyset$

$$\Pi(L, d_1 \text{ alt } d_2) \xrightarrow{\tau_{alt}} \Pi(L, d_1)$$ if $ll.(d_1 \text{ alt } d_2) \cap L \neq \emptyset$ for $i \in \{1, 2\}$

$$\Pi(ll.d_1 \cap L, d_1) \xrightarrow{\alpha} \Pi(ll.d_1 \cap L, d'_1)$$ if $ll.d_1 \cap L \neq \emptyset$

$$\Pi(L, d_1 \text{ seq } d_2) \xrightarrow{\alpha} \Pi(L, d'_1 \text{ seq } d_2)$$

$$\Pi(L, d_1 \text{ seq } d_2) \xrightarrow{\alpha} \Pi(L, d_1 \text{ seq } d'_2)$$

$$\Pi(L, d_1) \xrightarrow{\alpha} \Pi(L, d'_1)$$ if $d_1 \equiv d_2$ and $d'_1 \equiv d'_2$

$$\Pi(L, d_2) \xrightarrow{\alpha} \Pi(L, d'_2)$$

$$\Pi(ll.d_2 \cap L, d_2) \xrightarrow{\alpha} \Pi(ll.d_2 \cap L, d'_2)$$

$$\Pi(L, \text{ loop}(0..*) (d)) \xrightarrow{\tau_{loop}} \Pi(L, \text{ skip alt (d seq loop}(0..*) (d))))$$

For more explanation of the rules of the projection system, the reader is referred to [12].

The projection system of Def. 4 is based on [12] where parameter variables are not taken into consideration. Recall that each parameter variable is bound to a new value upon the occurrence of the event it is contained in. This has the consequence that parameter variables occurring inside a loop are bound to new values for each iteration of the loop. Thus to modify the projection system of Def. 4 to take this into account, we only need to modify the rule for loop(0..*) (the last rule of Def. 4). To simulate the fact that parameter variables are bound to new values in each iteration of the loop, we let the projection system rename all parameter variables by incrementing their index for each iteration of the loop. Formally, we make use of the function $ipv(\_ ) \in \text{PVar} \rightarrow \text{PVar}$ that increments the index of a parameter variable by one, i.e.,

$$ipv((vn, i)) = (vn, i + 1)$$

The function is lifted to diagrams such that $ipv(d)$ yields the diagram obtained from $d$ by incrementing all its parameter variables by one. The revised projection system is now given by the following definition.

**Definition 5 (Revised projection system)** The revised projection system that handles parameter variables is the LTS over

$$\alpha \tau \in \{\tau_{refuse}, \tau_{alt}, \tau_{loop}\} \cup \text{E} \cup \text{C} \cup \text{A}$$

whose states are

$$\Pi'(l.\_\_ \_ \_ ) \in \text{P}(L) \times \text{D}$$

and whose transitions are exactly those that can be derived by the rules of Def. 4 except for rule for loop(0..*) which is redefined as follows:

$$\Pi'(L, \text{ loop}(0..*) (d)) \xrightarrow{\tau_{loop}} \Pi'(L, \text{ skip alt (d seq loop}(0..*) (ipv(d))))$$
A.2.2 Evaluation and data states

In order to define the operational semantics of sequence diagrams, we need to describe how the data states change throughout execution. In this section, we present some auxiliary functions that are needed for this purpose.

An expression \( ex \in \text{Exp} \) is closed if \( \text{var}(ex) = \emptyset \). We let \( C\text{Exp} \) denote the set of closed expressions, defined as:

\[
C\text{Exp} \overset{\text{def}}{=} \{ ex \in \text{Exp} \mid \text{var}(ex) = \emptyset \}
\]

We assume the existence of a function \( \text{eval} \in C\text{Exp} \to \text{Val} \cup \{ \perp \} \) that evaluates any closed expression to its value. If an expression \( ex \) is not well formed or otherwise cannot be evaluated (e.g., because of division by zero), then \( \text{eval}(ex) = \perp \).

The evaluation function is lifted to signals, messages, and events such that \( \text{eval}(si), \text{eval}(m), \text{eval}(e) \) evaluate all expressions of signal \( si \), message \( m \), and event \( e \), respectively. For example, we have that

\[
\text{eval}(\text{msg}(1 + 2, 4 - 1)) = \text{msg}(\text{eval}(1 + 2), \text{eval}(4 - 1)) = \text{msg}(3, 3)
\]

If an expression \( ex \) in signal \( si \) is not well formed, i.e., \( \text{eval}(ex) = \perp \), then \( \text{eval}(si) = \perp \). If \( e \) is an event \((k, m)\) and \( si \) the signal of \( m \), then we also have that \( \text{eval}(m) = \perp \) and \( \text{eval}(e) = \perp \).

Let \( \sigma \in \text{Var} \to \text{Exp} \) be a mapping from variables to expressions. We denote such a mapping \( \sigma = \{ x_1 \mapsto ex_1, x_2 \mapsto ex_2, \dots, x_n \mapsto ex_n \} \) for distinct \( x_1, x_2, \dots, x_n \in \text{Var} \) and for \( ex_1, ex_2, \dots, ex_n \in \text{Exp} \). If \( ex_1, ex_2, \dots, ex_n \in \text{Val} \)

we call it a data state. We let \( \Sigma \) denote the set of all mappings and \( \hat{\Sigma} \) denote the set of all data states. We use the same convention for the set of all events \( \text{E} \), and denote by \( \hat{\text{E}} \) the set of all events whose signals have only values as arguments.

The empty mapping is denoted by \( \emptyset \). \( \text{Dom}(\sigma) \) denotes the domain of \( \sigma \), i.e.,

\[
\text{Dom}(\{ x_1 \mapsto ex_1, x_2 \mapsto ex_2, \dots, x_n \mapsto ex_n \}) \overset{\text{def}}{=} \{ x_1, x_2, \dots, x_n \}
\]

We let \( \sigma[x \mapsto ex] \) denote the mapping \( \sigma \) except that it maps \( x \) to \( ex \), i.e.,

\[
\{ x_1 \mapsto ex_1, \dots, x_n \mapsto ex_n \}[x \mapsto ex] \overset{\text{def}}{=} \begin{cases} 
\{ x_1 \mapsto ex_1, \dots, x_n \mapsto ex_n, x \mapsto ex \} & \text{if } x \neq x_i \text{ for all } i \in \{1, \ldots, n\} \\
\{ x_1 \mapsto ex_1, \dots, x_i \mapsto ex, \dots, x_n \mapsto ex_n \} & \text{if } x = x_i \text{ for some } i \in \{1, \ldots, n\}
\end{cases}
\]

We generalize \( \sigma[x \mapsto ex] \) to \( \sigma[\sigma'] \) in the following way:

\[
\sigma[\{ x_1 \mapsto ex_1, \ldots, x_n \mapsto ex_n \}] \overset{\text{def}}{=} \sigma[x_1 \mapsto ex_1] \cdots [x_n \mapsto ex_n]
\]

The mapping is lifted to expressions such that \( \sigma(ex) \) yields the expression obtained from \( ex \) by simultaneously substituting the variables of \( ex \) with the expressions that these variables map to in \( \sigma \). For example, we have that \( \{ y \mapsto 1, z \mapsto 2 \}(y + z) = 1 + 2 \). We furthermore lift \( \sigma \) to signals, messages, and events such that \( \sigma(si), \sigma(m), \sigma(e) \) yields the signal, message, and event obtained from \( si, m, \) and \( e \), respectively, by substituting the variables of their expressions according to \( \sigma \).
A.2.3 Execution system and trace semantics for sequence diagrams

The execution system of the operational semantics tells us how to execute a sequence diagram in a step by step manner. Unlike the projection system, the execution system keeps track of the communication medium and data states in addition to the diagram state. Thus a state of the execution system is a triple consisting of a communication medium, diagram, and data state:

\[ \text{AXS} \overset{\text{def}}{=} B \times D \times \hat{\Sigma}_T \]

Here \( \hat{\Sigma}_T \) denotes the set of total data states, i.e., the set of all data states \( \sigma \) satisfying

\[ \text{Dom}(\sigma) = \text{Var} \]

We assume a communication model where each message has its own channel from the transmitter to the receiver, something that allows for message overtaking. The communication medium keeps track of messages that are sent between lifelines of a diagram, i.e., the messages of transmission events are put into the communication medium, while the messages of receive events are removed from the communication medium.

It is only necessary to keep track of the communication between those lifelines that are present in a sequence diagram; messages received from the environment (i.e., from lifelines not present in a diagram) are always assumed to be enabled.

The states of the communication medium are of the form \((M, L)\) where \(M\) is a set of messages and \(L\) is a set of lifelines under consideration, i.e., the lifelines that are not part of the environment. The set of all communication medium states \(B\) is defined by

\[ B \overset{\text{def}}{=} \mathbb{P}(M) \times \mathbb{P}(L) \]

We define two functions for manipulating the communication medium: \(\text{add}, \text{rm} \in B \times M \rightarrow B\). The function \(\text{add}(\beta, m)\) adds the message \(m\) to the communication medium \(\beta\), while \(\text{rm}(\beta, m)\) removes the message \(m\) from the communication medium \(\beta\). We also define the predicate \(\text{ready} \in B \times M \rightarrow \text{Bool}\) that for a communication medium \(\beta\) and a message \(m\) yields true if \(\beta\) is in a state where it can deliver \(m\), and false otherwise. Formally, we have

\[
\begin{align*}
\text{add}((M, L), m) & \overset{\text{def}}{=} (M \cup \{m\}, L) \\
\text{rm}((M, L), m) & \overset{\text{def}}{=} (M \setminus \{m\}, L) \\
\text{ready}((M, L), m) & \overset{\text{def}}{=} \text{tr.m} \notin L \lor m \in M
\end{align*}
\]

We are now ready to define the execution system for sequence diagrams.

**Definition 6 (Execution system)** The execution system is an LTS whose states are

\[ B \times D \times \hat{\Sigma}_T \]

whose labels are

\[ \{\tau_{\text{refuse}}, \tau_{\text{alt}}, \tau_{\text{loop}}, \tau_{\text{assign}}, t, f, \bot\} \cup E \]

and whose transitions are exactly those that can be derived from the following rules

\[
\begin{align*}
\Pi'(l.l.d, d) \xrightarrow{\tau} \Pi'(l.l.d', d') & \text{ for } \tau \in \{\tau_{\text{refuse}}, \tau_{\text{loop}}, \tau_{\text{alt}}\} \\
[\beta, d, \sigma] \xrightarrow{\tau} [\beta, d', \sigma]
\end{align*}
\]
A UML SEQUENCE DIAGRAMS

See [12] for more details on the rules of the execution system.

The trace semantics of a sequence diagram is a pair consisting of a positive trace set and a negative trace set. The traces of a diagram \( d \) are obtained by recording the events occurring on the transitions of the execution system when executing \( d \) until it is reduced to a \texttt{skip} (which means that the diagram cannot be further executed).

To distinguish negative from positive traces, we make use of the silent event \( \tau_{\text{refuse}} \). That is, if a transition labeled by \( \tau_{\text{refuse}} \) is taken during execution, then this means that a negative trace is being recorded. Otherwise the trace is positive.

**Definition 7 (Trace semantics)** The trace semantics of \( d \), written \( [d] \), is then defined by

\[
[d] ≜ \left\{ s \mid \exists \beta ∈ \mathbb{B} : \exists \sigma, \sigma' ∈ \tilde{\Sigma}_T : \\
[(\emptyset, ll.i, d), d, \sigma] \xrightarrow{\text{assign}(x, ex, l)} [(\beta, skip, \sigma') ∧ s\{\tau_{\text{refuse}}, \bot\} = \emptyset], \\
\left\{ s | s \in E^+ \} \right. \right. \left. \left. \exists \beta ∈ \mathbb{B} : \exists \sigma, \sigma' ∈ \tilde{\Sigma}_T : \\
[(\emptyset, ll.i, d), d, \sigma] \xrightarrow{\text{assign}(x, ex, l)} [(\beta, skip, \sigma') ∧ s\{\tau_{\text{refuse}}, \bot\} \in \{s\{\text{reuse}\}^+\}]]
\]

Note that the projection function \( | \) takes a set \( A \) and a sequence \( s \) and yields the sequence \( s|_A \) obtained from \( s \) by removing all elements not in \( A \). Note also that \( A^+ \) denotes the set of sequences of \( A \) with at least one element, i.e., \( A^+ = A^* \setminus \{\emptyset\} \).

**A.3 Policy adherence for sequence diagrams**

In this section, we define what it means for a system to adhere to a policy expressed by a sequence diagram.

A system (interpreted as a set of traces of events) adheres to a sequence diagram policy if none of the traces of the system has a negative trace of a lifeline in the sequence diagram as a sub-trace. A trace \( s = \langle e_1, \ldots, e_n \rangle \) is a sub-trace of \( t \), written \( s ⊆ t \), iff

\[
s_1 ⊆ \langle e_1 \rangle \cdots ⊆ s_n ⊆ \langle e_n \rangle \cdots ⊆ s_{n+1} = t
\]
for some $s_1, \ldots, s_{n+1} \in E^*$. See [18] for a more precise definition.

We first formally define adherence for diagrams consisting of a single lifeline.

**Definition 8 (Policy adherence of single lifeline sequence diagrams)** Let $d$ be a single lifeline diagram, i.e., $d \in D_l$ for some lifeline $l$, and let $\Phi$ denote the traces of a system. Then the system adheres to the policy $d$, written $d \rightarrow_{da} \Phi$, iff

$$\left( s \in H_{neg} \wedge t \in \Phi\mid_{E_l} \right) \implies \neg(s \prec t) \text{ for } [d] = (H_{pos}, H_{neg})$$

Note that the projection operator $\mid_{E_l}$ is lifted to sets of sequences such that $\Phi\mid_{A}$ yields the set obtained from $\Phi$ by projecting each sequence of $\Phi$ to $A$, i.e., $\Phi\mid_{A} = \{s\mid_{A} \mid s \in \Phi\}$.

Adherence for general sequence diagrams (i.e., sequence diagrams that may contain more than one lifeline) is captured by the following definition.

**Definition 9 (Policy adherence of sequence diagrams)** Let $d$ be a sequence diagram, i.e., $d \in D$ and let $\Phi$ denote the traces of a system. Then the system adheres to the policy $d$, written $d \rightarrow_{dag} \Phi$, iff

$$\pi_l(d) \rightarrow_{da} \Phi \text{ for all } l \in ll.d$$

## B State machines

In this section, we define the syntax and the semantics of UML inspired state machines. We also define what it means for a system to adhere to a policy expressed as a state machine.

### B.1 Syntax

As illustrated in Fig. 11, the constructs which are used for specifying state machines are *initial state*, *simple state*, *final state*, *transition*, and *action expression*.

A state describes a period of time during the life of a state machine. The three kinds of states, *initial state, simple states*, and *final states*, are graphically represented by a black circle, a box with rounded edges, and a black circle encapsulated by another circle, respectively.
A transition represents a move from one state to another. In the graphical diagrams, transitions are labeled by action expressions of the form

\[ nm.si[bx]/ef \]

Here the expression \( nm.si \) where \( nm \) is a state machine name and \( si \) is a signal is called an event trigger. The expression \( [bx] \) where \( bx \) is a boolean expression is called a guard, and \( ef \) is called an effect. Intuitively, the action should be understood as follows: when signal \( si \) is received from a state machine with name \( nm \) and the boolean expression \( bx \) evaluates to true, then the effect \( ef \) is executed. An effect is a sequence of assignments and/or an output expression of the form \( nm.si \) representing the transmission of signal \( si \) to the state machine with name \( nm \).

We will henceforth consider action expressions that contain at most one event. In our formal representation of state machines, we will therefore use action expressions of the form \((e, bx, sa)\) where \( e \) is an input or output event, \( bx \) is a boolean expression (the guard) and \( sa \) is a sequence of assignments of the form \(((x_1, e x_1), \ldots, (x_n, e x_n))\). Formally, the set of all action expressions w.r.t. the set of events \( E \) is defined by

\[
\text{Act}_E \overset{\text{def}}{=} (E \cup \{\epsilon\}) \times \text{BExp} \times (\text{Var} \times \text{Exp})^* 
\]

Note that the event is optional in an action. An action without an event is of the form \((\epsilon, bx, sa)\). We will henceforth use \( e_\epsilon \) to denote an arbitrary event or an empty expression, i.e., \( e_\epsilon \) denotes a member of \( E \cup \{\epsilon\} \).

The alphabet of a state machine is a set of events containing signals whose arguments are distinct parameter variables. We require that all events in the alphabet are distinct when two events \( e \) and \( e' \) are considered equal if they have the same name and the same number of arguments.

To make this more precise, we let \( E_{pv} \) denote the set of all events whose signals contain distinct parameter variables only, i.e.,

\[
\forall (k, (nm_t, nm_r, st(ex_1, \ldots, ex_n))) \in E : \\
\land e_1 \in \text{PVar} \land \cdots \land e_n \in \text{PVar} \\
\land \forall i, j \in \{1, \ldots, n\} : \\
i \neq j \implies e_i \neq e_j \\
\iff (k, (nm_t, nm_r, st(ex_1, \ldots, ex_n))) \in E_{pv} 
\]

Note that the formula is written in a style suggested by Lamport [10]. Here, the arguments of a conjunction may be written as an aligned list where \( \land \) is the first symbol before each argument. A similar convention is used for disjunctions. Also, indentation is sometimes used instead of parentheses.

We write \( e = e' \) if events \( e \) and \( e' \) have the same kind, transmitter, and receiver and their signals have the same name and the same numbers of arguments, i.e.,

\[
(k, (nm_t, nm_r, st(ex_1, \ldots, ex_j))) = (k', (nm'_t, nm'_r, st'(ex'_1, \ldots, ex'_k))) \\
\iff k = k' \land nm_t = nm'_t \land nm_r = nm'_r \land st = st' \land j = k
\]

We are now ready to define the syntax of state machines precisely.

**Definition 10 (State machines)** A state machine is a tuple \((E, Q, R, q_I, F)\) consisting of
• an alphabet $\mathcal{E} \subseteq E_{pv}$ where $e, e' \in \mathcal{E}$ \implies \neg(e = e')
• a set of states $Q$
• a transition relation $R \subseteq Q \times \text{Act}_{\mathcal{E}} \times Q$
• an initial state $q_I \in Q$
• a set of final states $\mathcal{F} \subseteq Q$

The set of all state machines is denoted by $\text{SM}$.

We define the functions for obtaining the alphabet, states, transitions, initial state, and final states of a state machine:

\[
\begin{align*}
alph((\mathcal{E}, Q, R, q_I, F)) & \triangleq \mathcal{E} \\
states((\mathcal{E}, Q, R, q_I, F)) & \triangleq Q \\
trans((\mathcal{E}, Q, R, q_I, F)) & \triangleq R \\
init((\mathcal{E}, Q, R, q_I, F)) & \triangleq q_I \\
final((\mathcal{E}, Q, R, q_I, F)) & \triangleq \mathcal{F}
\end{align*}
\]

B.1.1 Syntax constraints

We impose one restriction on the set of syntactically correct action expressions $\text{Act}_{\mathcal{E}}$: the parameter variables of the guard and the assignment sequence must be a subset of the parameter variables of the event (if the event is present in the action):

\[
(e, bx, sa) \in \text{Act}_{\mathcal{E}} \implies (pvar(bx) \cup pvar(sa)) \subseteq pvar(e) \tag{23}
\]

where $pvar$ yields the set of all parameter variables in an expression, assignment sequence, or an event.

We define four syntax rules for state machines, and we say that a state machine $SM$ is well formed if it satisfies these rules:

**SM1** The initial state of $SM$ has zero ingoing transitions.

**SM2** The initial state of $SM$ has exactly one outgoing transition, and the action expression of this transition does not contain an event or a guard.

**SM3** Each transition of $SM$ (except the initial transition) is labeled by an action expression that contains an event.

**SM4** All variables (except parameter variables) in $SM$ must be explicitly assigned to a value before they are used.

B.2 Semantics

In this section, we define the semantics of state machines. First we define the execution graph of a state machine, then we define the traces obtained by executing a state machine.

The execution graph of a state machine $SM$ is an LTS whose states are pairs $[q, \sigma]$ where $q$ is a state of $SM$ and $\sigma$ is a data state. The transitions of the execution graph are defined in terms of the transitions of $SM$. That is, if
SM has a transition from $q$ to $q'$ that is labeled by $(e, bx, sa)$ and the signal of $e$ has no arguments, then the execution graph has a transition from $[q, \sigma]$ to $[q', \sigma']$ that is labeled by $e$ provided that the guard $bx$ is evaluated to true under state $\sigma$. Here, the data state $\sigma'$ is equal to $\sigma$ except that the variables of $sa$ are assigned to new values as specified by $sa$. To make this precise, we define the function $\text{as2ds} : \hat{\Sigma} \times (\text{Var} \times \text{Exp})^* \rightarrow \hat{\Sigma}$ which takes a data state $\sigma$ and an assignment sequence $sa$ and yields a new updated data state. Formally,

$$\text{as2ds}(\sigma, (x, ex)) \overset{\text{def}}{=} \sigma$$

$$\text{as2ds}(\sigma, (x, ex) \sim sa) \overset{\text{def}}{=} \text{as2ds}(\sigma[x \mapsto eval(\sigma(ex))], sa)$$

If the signal of event $e$ contains arguments, i.e., parameter variables, then these variables are bound the new arbitrary values. In this case, the guard $bx$ and the event $e$ are evaluated under some data state $\sigma''$ where $\sigma''$ is an arbitrary mapping whose domain equals the parameter variables of $e$.

**Definition 11 (Execution graph of state machines)** The execution graph of state machine $SM = (E, Q, R, q_I, F)$, written $\text{EG}(SM)$, is the LTS over $\{\epsilon\} \cup E$ whose states are $Q \times \hat{\Sigma}_T$ and whose transitions are exactly those that can be derived from the following rule:

$$q \xrightarrow{(e, bx, sa)} q' \in R$$

$$[q, \sigma] \xrightarrow{\text{eval}(\sigma[\sigma'')(bx)) = t \land \text{Dom}(\sigma') = \text{pvar}(e, \sigma'' \text{as2ds}([\sigma], sa))} [q', \text{as2ds}([\sigma], sa)]$$

The trace semantics of a state machine is the set of sequences obtained by recording the events occurring in each path from the initial state to a final state of the state machine.

**Definition 12 (Trace semantics of state machines)** The trace semantics of a state machine $SM = (E, Q, R, q_I, F)$, written $\llbracket SM \rrbracket$, is defined by

$$\llbracket SM \rrbracket \overset{\text{def}}{=} \{s \mid \exists q' \in F \exists q, \sigma' \in \hat{\Sigma}_T : [q_I, \sigma] \xrightarrow{s} [q', \sigma'] \in \text{EG}(SM)\}$$

### B.3 Policy adherence for state machines

In this section, we define what it means for a system to adhere to a policy expressed as a state machine. We also define what it means for a system to adhere to a set of state machines.

Intuitively, a system $S$ adheres to a state machine policy $SM$ if all execution traces of $S$ (when restricted to the alphabet of $SM$) are described by $SM$. This is formally captured by the following definition.

**Definition 13 (Policy adherence for a state machine)** Let $SM$ be a state machine defining a policy and let $\Phi$ denote the traces of a system. Then the system adheres to $SM$, written $SM \rightarrow_{sa} \Phi$, iff

$$\Phi|_E \subseteq \llbracket SM \rrbracket$$

where $E \overset{\text{def}}{=} \{e \in \hat{E} | e' \in \text{alph}(SM) \land e = e'\}$. 


Adherence for a set of state machines is precisely captured by the following definition.

**Definition 14 (Policy adherence for a set of state machines)** Let $SMS$ be a set of state machines and let $\Phi$ denote the traces of a system. Then the system adheres to $SMS$, written $SMS \rightarrow_{sa} \Phi$, iff

$$SM \rightarrow_{sa} \Phi \quad \text{for all } SM \in SMS$$

### C Specifying transformations using sequence diagrams

In this section, we show how transformations can be expressed in terms of sequence diagrams.

#### C.1 Transformation specifications

A transformation specification is a set of mapping rules. A mapping rule is a pair $(dp, dp')$ consisting of a left hand side sequence diagram pattern $dp$ and a right hand side sequence diagram pattern $dp'$. A sequence diagram pattern is a sequence diagram whose atoms (events, constraints, and assignments) may contain meta variables.

We let $MVar$ denote the set of all meta variables and we let $mv$ range over this set. Events that may contain meta variables are called event patterns. The set $EP$ of all event patterns is defined

$$EP \triangleq K \times ((L \cup MVar) \times (L \cup MVar) \times (SIP \cup MVar))$$

Here $SIP$ denotes the set of all signal patterns. This set is defined by

$$SIP \triangleq (NM \cup MVar) \times (ExpP)^*$$

where $ExpP$ is an expression that may contain meta variables (in addition to normal variables and parameter variables).

A constraint pattern is an expression of the form

$$\text{constr}(bxp, l)$$

where $bxp$ is a boolean expression that may contain meta variables. We let $CP$ denote all constraint patterns.

An assignment pattern is an expression of the form

$$\text{assign}(x, exp, l)$$

where $exp$ is an expression that may contain meta variables. We let $AP$ denote the set of all assignment patterns.

**Definition 15 (Sequence diagram pattern)** The set of sequence diagram patterns $DP$ is defined by the following syntax

$$dp ::= mv | ep | cp | ap | \text{refuse}(dp) | \text{loop}(0..*)(dp) |
\text{refuse}(dp) | \text{loop}(0..*)(dp) |
\text{dp}_1 \text{ seq } \text{dp}_2 | \text{dp}_1 \text{ alt } \text{dp}_2 | \text{dp}_1 \text{ par } \text{dp}_2$$

A sequence diagram pattern is either a meta variable ($mv$), an event pattern ($ep$), a constraint pattern ($cp$), an assignment pattern ($ap$), or the composition of one or more diagram patterns.
C.2 Transformation

In this section, we define the function induced by a transformation specification. Intuitively, when a transformation specification \( ts \) is applied to a sequence diagram \( d \), all fragments of \( d \) that match a left hand side pattern of a mapping rule in \( ts \) are replaced by the right hand side pattern of the mapping rule.

Matching is defined in terms of a substitution \( sub \in \text{MVar} \rightarrow (\mathcal{D} \cup \text{Exp}) \) that replaces meta variables by diagrams or expressions. Any substitution \( sub \) is lifted to diagram patterns such that \( sub(dp) \) yields the diagram obtained from \( dp \) by simultaneously replacing all meta variables in \( dp \) by diagrams or expressions according to \( sub \). The set of all substitution is denoted by \( \text{Sub} \).

A diagram pattern \( dp \) matches a diagram \( d \) if there is a substitution \( sub \) such that \( sub(dp) = d \).

We say that the domain of a mapping rule \( (dp, dp') \), written \( \text{Dom}((dp, dp')) \), is the set of all diagrams that can be matched by its left hand side pattern, i.e.,

\[
\text{Dom}((dp, dp')) = \{d \in \mathcal{D} : \exists sub \in \text{Sub} : sub(dp) = d\}
\]

To ensure that transformation specifications induce functional transformations, we require that the mapping rules of a transformation specification must have disjoint domains, i.e., each transformation specification \( ts \) must satisfy the following constraint

\[
\forall (dp_1, dp'_1) \in ts : \forall (dp_2, dp'_2) \in ts : (dp_1, dp'_1) \neq (dp_2, dp'_2) \implies \text{Dom}((dp_1, dp'_1)) \cap \text{Dom}((dp_2, dp'_2)) = \emptyset
\]

**Definition 16 (Function induced by a transformation specification)** The function \( T_{ts} \in \mathcal{D} \rightarrow \mathcal{D} \) induced by transformation specification \( ts \) is defined as follows

\[
\text{if } sub(dp) = d \text{ for some } (dp, dp') \in ts \text{ and } sub \in \text{Sub} \\
\text{then } T_{ts}(d) = sub(dp') \\
\text{else if } d = d_1 \text{ op } d_2 \text{ for some } d_1, d_2 \in \mathcal{D} \text{ and } \text{op} \in \{\text{seq}, \text{alt}, \text{par}\} \\
\text{then } T_{ts}(d) = T_{ts}(d_1) \text{ op } T_{ts}(d_2) \\
\text{else if } d = \text{op}(d_1) \text{ for some } d_1 \in \mathcal{D} \text{ and } \text{op} \in \{\text{loop(0..*)}, \text{refuse}\} \\
\text{then } T_{ts}(d) = \text{op}(T_{ts}(d)) \\
\text{else} \\
T_{ts}(d) = d
\]

D From sequence diagrams to state machines

In this section, we define the transformation from sequence diagrams to state machines. In general, the transformation of a sequence diagram yields a set of state machines, i.e., one state machine for each lifeline in the sequence diagram.

The main requirement to the transformation is that is should be adherence preserving.
Definition 17 (Adherence preservation) Let $T \in D \rightarrow \mathcal{P}(SM)$ be a transformation from sequence diagrams to sets of state machines. Then $T$ is adherence preserving if for every system with traces $\Phi$ and sequence diagram policy $d$, the system adheres to $d$ if and only if it adheres to $T(d)$, i.e.,

$$d \xrightarrow{\text{dag}} \Phi \iff T(d) \xrightarrow{\text{sag}} \Phi$$

We first, in Sect. D.1, define the transformation from a sequence diagram with only one lifeline to a state machine. Then, in Sect. D.2, we define the transformation from (general) sequence diagrams to state machine sets. We show that this transformation is adherence preserving for policies that are composed of sub-policies with disjoint sets of variables.

D.1 From single lifeline diagrams to state machines

The transformation from a single lifeline diagram $d$ to a state machine has two phases. In phase 1, the sequence diagram $d$ is transformed into a state machine $SM$ whose trace semantics equals the negative trace set of $[d]$. In phase 2, $SM$ is inverted into the state machine $SM'$ whose semantics is the set of all traces that do not have a trace of $SM$ as a sub-trace.

Definition 18 (Single lifeline sequence diagram to basic state machine)
The transformation $d2p \in D \rightarrow SM$ from single lifeline diagrams to state machines is defined by

$$d2p \overset{\text{def}}{=} ph2 \circ ph1$$

where $ph1$ and $ph2$ represent phase 1 and 2 (as defined below).

D.1.1 Phase 1

In phase 1, the sequence diagram $d$ is transformed into a state machine $SM$ that describes the negative traces of $d$. The state machine $SM$ corresponds to the projection system induced by $d$. That is, if the projection system has a transition $\Pi(ll.d,d') \xrightarrow{e} \Pi(ll.d,d'')$, then $SM$ has a transition $q \xrightarrow{e} q''$ where $q$ and $q'$ correspond to $\Pi(ll.d,d)$ and $\Pi(ll.d,d')$, respectively. However, some transitions of the projection system are truncated. In particular,

- all silent events are removed, e.g., if $\Pi(ll.d,d) \xrightarrow{\text{silent}} \Pi(ll.d,d') \xrightarrow{e} \Pi(ll.d,d'')$, then $SM$ has a transition $q \xrightarrow{e} q''$;

- constraints are concatenated with succeeding events and assignments concatenated with preceding events, e.g., if $\Pi(ll.d,d) \xrightarrow{\text{constr}(bx,l)} \Pi(ll.d,d1) \xrightarrow{e} \Pi(ll.d,d2) \xrightarrow{\text{assign}(x,ex,l)} \Pi(ll.d,d3)$, then $SM$ has a transition $q \xrightarrow{(e,bx,(x,ex))} q3$.

To define this precisely, we introduce the notion of experiment relation.

Definition 19 (Experiment relations) The relations $\Rightarrow$, $\Rightarrow$, and $\Rightarrow$ for any $\alpha \in (E \cup C \cup A)^*$ and $s \in (E \cup C \cup A)^*$ are defined as follows

1. $q \Rightarrow q'$ means that there is a sequence of zero or more transitions from $q$ to $q'$ that are labeled by silent events, i.e., $q \xrightarrow{\tau_1,\ldots,\tau_n} q'$ for $\tau_1,\ldots,\tau_n \in \{\text{alt},\text{refuse},\text{loop}\}$.
2. \( q \Rightarrow q' \) means that \( q \Rightarrow q_1 \xrightarrow{\tau_{\text{refuse}}} q_2 \Rightarrow q' \) for some states \( q_1 \) and \( q_2 \);

3. if \( s = (q_1, q_2, \ldots, q_n) \), then \( q \Rightarrow q' \) means that \( q \Rightarrow q_1 \xrightarrow{\tau_{\text{refuse}}} q_2 \cdots \xrightarrow{\tau_{\text{refuse}}} q' \).

To obtain correct action expressions for transitions, we define the function \( c2g : C^* \rightarrow \text{BExp} \) for converting a sequence of constraints into a guard and the function \( a2ef : A^* \rightarrow (\text{Var} \times \text{Exp})^* \) for converting a sequence of sequence diagram assignments into a sequence of state machine assignments. More precisely, these functions are defined as follows

\[
\begin{align*}
c2g((\text{constr}(b_1, 1), \ldots, \text{constr}(b_n, l))) & \overset{\text{def}}{=} \text{conj}(b_1, \ldots, b_n) \\
a2ef((\text{assign}(x_1, e_1, l), \ldots, \text{assign}(x_n, e_n, l))) & \overset{\text{def}}{=} ((x_1, e_1), \ldots, (x_n, e_n))
\end{align*}
\]

where \( \text{conj} : \text{BExp}^* \rightarrow \text{BExp} \) yields the conjunction of a sequence of boolean expressions. For the empty sequence, \( \text{conj} \) yields true, i.e., \( \text{conj}() = t \). We use the function \( \text{conj} \) instead of expressing the conjunction directly because we have not defined the notation for boolean expressions in \( \text{BExp} \) since this is not important in this report.

To distinguish negative from positive traces, we make use of the \( \tau_{\text{refuse}} \) silent event. That is, any execution that involves a \( \tau_{\text{refuse}} \) represents a negative behavior. Otherwise the execution represents positive behavior.

**Definition 20 (Positive and negative experiment relations)** The relations \( \Rightarrow_{\text{pos}} \) and \( \Rightarrow_{\text{neg}} \) for any \( s \in (E \cup C \cup A)^* \) are defined as follows

1. \( q \Rightarrow_{\text{neg}} q' \) means that \( q \Rightarrow q_1 \xrightarrow{\tau_{\text{refuse}}} q_2 \Rightarrow q' \) for some states \( q_1 \) and \( q_2 \)
   and some traces \( t \) and \( u \) such that \( s = t \xrightarrow{\tau_{\text{refuse}}} u \)
2. \( q \Rightarrow_{\text{pos}} q' \) means that \( q \Rightarrow q' \) and not \( q \Rightarrow_{\text{neg}} q' \)

Since the goal of phase 1 is to construct a state machine \( SM \) that describes the negative traces of a sequence diagram, each final state of \( SM \) should accept a negative trace. To distinguish these final states from those that accept positive traces, we let each state of \( SM \) have one of two modes: \( \text{pos} \) and \( \text{neg} \). If a state has mode \( \text{pos} \), then this means that a positive trace is being recorded when this state is entered. If a state has mode \( \text{neg} \), then a negative trace is being recorded when the state is entered.

Even though we shall restrict attention to well formed sequence diagrams, we cannot in general let the alphabet of the state machine be equal to the set of all events in the sequence diagram, because this set may not satisfy the requirement that all events in the alphabet must be distinct up to argument renaming (see Def. 10). To ensure that a correct alphabet is constructed, we make use of the function \( \psi \in \text{PVar} \rightarrow \text{PVar} \) that renames parameter variables. We lift the function to signals with parameter variables as arguments as follows:

\[
\psi(st(pv_1, pv_2, \ldots, pv_n)) \overset{\text{def}}{=} st(\psi(pv_1), \psi(pv_2), \ldots, \psi(pv_n))
\]

To ensure that the renaming function does not change the meaning of a signal, we require that \( \psi \) does not rename two distinct parameter variables of a signal into the same parameter variable, i.e., we require

\[
\forall i, j \in \{1, \ldots, n\} : (\psi(st(pv_1, \ldots, pv_n))) = st(pv'_1, \ldots, pv'_n) \land pv_i \neq pv_j \implies pv'_i \neq pv'_j
\]
Figure 12: State machines $W$ and $W'$ are obtained by transformation without and with condition \textit{Last}, respectively.

We lift $\psi$ to expressions, events, and actions in the obvious way. Furthermore, we lift $\psi$ to event sets and transition sets as follows:

\[
\psi(E) \overset{\text{def}}{=} \{ \psi(e) \in E_{\text{pv}} \mid e \in E \}
\]

\[
\psi(R) \overset{\text{def}}{=} \{ q \xrightarrow{\psi(\text{act})} q' \mid q \xrightarrow{\text{act}} q' \in R \}
\]

We are now ready to define the transformation of phase 1.

\textbf{Definition 21 (Phase 1)} The transformation $\phi_1 : D \rightarrow \text{SM}$ which takes a single lifeline sequence diagram $d$ and yields a state machine describing the negative traces of $d$ is defined by

\[
\phi_1(d) = (\psi(eca.d \cap E), Q, \psi(R), \{(d), \text{pos}\}, \{(Q, \text{neg}) \in Q \mid \text{skip} \in Q\})
\]

where

\[
Q = \mathcal{P}(D) \times \{\text{pos}, \text{neg}\}
\]

$\psi \in \mathcal{P}\text{Var} \rightarrow \mathcal{P}\text{Var}$ renames parameter variables such that

\[
\forall e, e' \in \psi(eca.d \cap E) : -(e = e')
\]
and transition relation $\mathcal{R}$ is defined by the following formula

$$
\text{let } \text{Last}(d') \equiv \forall e \in (E \cup C) : \exists d'' \in D : \Pi(ll.d, d') \xrightarrow{ec} \Pi(ll.d, d'')
$$

$$
\forall d' = \text{skip}
$$

$$
\text{St}(Q, t, mo) \equiv \{ d'' \in D | d' \in Q \land \Pi(ll.d, d') \xrightarrow{ta} \Pi(ll.d, d'') \land \text{Last}(d'') \}
$$

in

$$
\forall tc \in C^* : \forall e \in eca.d \cap E : \forall ta \in A^* :
\forall (Q, mo) \in Q : \forall mo' \in \{ \text{pos, neg} \} :
\forall St\{d\}, ta, mo' \neq \emptyset
\Rightarrow (\{d\}, \text{pos}) \xrightarrow{(c, e, ta)} (St\{d\}, ta, mo') \in \mathcal{R}
\land St(Q, tc \xrightarrow{e} ta, pos) \neq \emptyset
\
\Rightarrow (Q, mo) \xrightarrow{(c, e, ta, a2e(f(ta)))} (St(Q, tc \xrightarrow{e} ta, pos), mo) \in \mathcal{R}
\land St(Q, tc \xrightarrow{e} ta, neg) \neq \emptyset
\Rightarrow (Q, mo) \xrightarrow{(c, e, ta, a2e(f(ta)))} (St(Q, tc \xrightarrow{e} ta, neg), neg) \in \mathcal{R}
$$

The predicate $\text{Last}$ (in Def. 21) ensures that the longest possible sequence of assignments is selected. For instance, the condition ensures that the following sequence diagram

```
refuse (a seq assign(i = 0), l)
seq assign(j = 0), l)
```

is transformed into the state machine $W'$ in Fig. 12, and not the state machine $W$ of Fig. 12. Note that the projection system consisting of those states that can be reached from the sequence diagram is illustrated at the top of Fig. 12.

Each state of the state machine constructed in phase 1 consists of a set of diagrams $Q$ rather than a single diagram which is used by the projection system. This is to reduce nondeterminism in the constructed state machine. To see how this works, consider the LTS labeled $A$ illustrated on the left hand side of Fig. 13. If we convert this into a state machine by removing silent events without merging states, we would obtain the state machine $B$ shown in the middle of Fig. 13 (note that we have omitted to specify the modes of states in the figure). Clearly, this state machine is nondeterministic. However, if we merge states 3 and 4, we obtain the state machine $C$ (on the right hand side of Fig. 13) which is deterministic.

**Lemma 1** Let $d$ be a well formed single lifeline sequence diagram, then the state machine $\text{ph1}(d)$ describes the negative traces of $d$, i.e.,

$$
\llbracket \text{ph1}(d) \rrbracket = H_{\text{neg}} \quad \text{for } \llbracket d \rrbracket = (H_{\text{pos}}, H_{\text{neg}})
$$

**D.1.2 Phase 2**

In phase 2, the state machine obtained from phase 1 is inverted into a state machine $S'M'$ whose semantics is the set of all traces that do not have a trace of $SM$ as a sub-trace. This notion of inversion is captured by the following definition.

**Definition 22 (Inversion)** State machine $S'M'$ is an inversion of state machine $SM$, written $\text{inv}(SM, S'M')$, iff

$$
\alpha\text{ph}(SM) = \alpha\text{ph}(S'M')
$$
Figure 13: Machines A and B are nondeterministic while C is deterministic.

and for all \( s \in \{ e \in \tilde{E} | \exists e' \in \text{alph}(SM) : e = e' \}^* \)

\[ (\forall t \in \llbracket SM \rrbracket : - (t \circ s)) \iff s \in \llbracket SM' \rrbracket \]

To explain how the transformation of phase 2 works, we first define the transformation for state machines whose transitions each contain exactly one event (whose signal has zero arguments) and no guards or assignments.

**Definition 23 (Phase 2 - Preliminary definition 1)** The transformation \( ph^2 \in \text{SM} \rightarrow \text{SM} \) which yields the inversion of state machines whose transitions each contain exactly one event (whose signal has zero arguments) and no guards or assignments is defined by

\[
ph^2'((\mathcal{E}, \mathcal{Q}, \mathcal{R}, q_I, \mathcal{F})) \overset{def}{=} (\mathcal{E}, \mathcal{P}(\mathcal{Q}), \mathcal{R}', \{q_I\}, \mathcal{P}(\mathcal{Q}))
\]

where the transition relation \( \mathcal{R}' \) is defined by the formula

\[
\text{let } St(Q, e) \overset{def}{=} \{ q' \in \mathcal{Q} | \exists q \in \mathcal{Q} : q \xrightarrow{e} q' \in \mathcal{R} \}
\]

\[
\forall Q \in \mathcal{P}(\mathcal{Q}) : \\
\forall e \in \mathcal{E} : \\
St(Q, e) \cap \mathcal{F} = \emptyset \iff Q \xrightarrow{e} Q \cup St(Q, e) \in \mathcal{R}'
\]

The rule for generating transitions ensures that previously visited states are “recorded”. To see why this is needed, consider the state machine \( P \) on the left hand side of Fig. 14. The set of traces described by it is

\[
\{ \langle a, a \rangle, \langle b, b \rangle \}
\]

Initially, both \( a \) and \( b \) are enabled. However, if \( a \) has occurred, then only \( b \) is enabled (if we assume that the alphabet of the state machine is \( \{a, b\} \)). Similarly if \( b \) has occurred, then only \( a \) is enabled. If both \( a \) and \( b \) have occurred, then no events are enabled. The final states of \( P \) are used to find those events that should not be enabled in \( P' \). For instance, consider the transition \( \{1\} \xrightarrow{a} \{1,2\} \) in \( P' \). Here the state 1 is “collected” because we need to make sure that \( b \) is not enabled after \( b \) has occurred in state \( \{1,2\} \). Since 1 is collected, the occurrence
of b in state \( \{1,2\} \) leads to state \( \{1,2,3\} \) and b is not enabled in this state because the occurrence of b in state 3 leads to a final state in P.

The following lemma shows that the transformation of phase 2 is correct for simple state machines, i.e., state machines with no guards or assignments.

**Lemma 2** Let SM be a state machine whose transitions each contain exactly one event (whose signal has zero arguments) and no guards or assignments. Then \( \text{ph}^{2}(SM) \) is an inversion of SM if \( \langle \rangle \notin \llbracket SM \rrbracket \), i.e.,

\[
\text{inv}(SM, \text{ph}^{2}(SM))
\]

The transformation of phase 2 is more complicated for state machines whose transitions contain guards. To see this, consider the state machine Q depicted on the left hand side of Fig.15. Inverting this state machine according to the transformation of Def. 23 would not work because the transformation does not take the guards into consideration. A correct inversion of Q is given by the state machine Q’ depicted on the right hand side of Fig.15. Here we see that the transitions

\[
1 \xrightarrow{(a,bx_{1},r)} 2 \quad \text{and} \quad 1 \xrightarrow{(a,bx_{2},r)} 3
\]
of state machine $Q$ (where $bx_1$ denotes $i <= 10$ and $bx_2$ denotes $i = 10$) have been converted into the transitions

\[
\begin{align*}
\{1\} & \xrightarrow{(a, \text{not}(bx_1 \text{ or } bx_2), c)} \{1\} \\
\{1\} & \xrightarrow{(a, bx_1 \text{ and not}(bx_2), c)} \{1, 2\} \\
\{1\} & \xrightarrow{(a, bx_2, c)} \{1, 3\} \\
\end{align*}
\]

In general, if a state machine $SM$ has transitions

\[
\begin{align*}
q & \xrightarrow{(e, bx_1, sa_1)} q_1, q & \xrightarrow{(e, bx_1, sa_2)} q_2, \ldots, q & \xrightarrow{(e, bx_n, sa_n)} q_n
\end{align*}
\]

where $q_1, \ldots, q_n$ are not final states, and its inversion $SM'$ has a state $\{q\}$, then for each set of indexes $Ix \subseteq \{1, \ldots, n\}$, the inversion $SM'$ has a transition

\[
\begin{align*}
\{q\} & \xrightarrow{(e, bx, sa)} \{q\} \cup Q
\end{align*}
\]

where $bx$ is the conjunction of those guards $bx_i$ that have an index in $Ix$ (i.e., $i \in Ix$), $bx'$ is the negation of the disjunction of the guards that do not have an index in $Ix$, $sa$ is the concatenated sequence of assignment sequences $sa_i$ that have an index in $Ix$, and $Q$ is the set of states $q_i$ that have an index in $Ix$.

Note that for the special case where $Ix = \emptyset$, then $bx$ should be equal to true. In addition, for the special case where $Ix = \{1, \ldots, n\}$, then $bx'$ should be equal to true.

To make this more precise, we make use of the function $\oplus : \mathbb{P}(\mathbb{N}) \times A \rightarrow A$ (where $\mathbb{N}$ denotes the set of all natural numbers, and $A$ denotes the set of all sequences), that for a set of indexes $Ix$ and a sequence $s$, yields the sequence obtained from $s$ by removing all elements whose index is not in $Ix$, e.g.,

\[
\begin{align*}
\{1, 3, 6\} \oplus (a, b, c, d, e) & = (a, c) \\
\{2, 4, 5\} \oplus (a, b, c, d, e) & = (b, d, e)
\end{align*}
\]

In addition, we let $\text{list}$ be a function that turns a set into a list, $\text{set}$ be a function that turns a list into a set (according to some total ordering on the elements in the set), and $\text{flatten}$ be the function that flattens a nested sequence, e.g.,

\[
\begin{align*}
\text{list}((a, b, c)) & = (a, b, c) \\
\text{list}((b, a, c)) & = (a, b, c) \\
\text{set}((a, b, a, c)) & = \{a, b, c\} \\
\text{flatten}((a, (b, c), ()), f)) & = (a, b, c, f)
\end{align*}
\]

We also need functions on boolean expressions. As before, we let the function $\text{conj}$ yield the conjunction of a sequence of boolean expressions. We also define the function $\text{disj} \in \mathbf{BExp}^* \rightarrow \mathbf{BExp}$ which yields the disjunction of a sequence of boolean expressions. For the empty sequence, $\text{disj}$ yields false, i.e., $\text{disj}(())) = \mathbf{f}$. Finally, we let $\text{neg} \in \mathbf{BExp} \rightarrow \mathbf{BExp}$ be the function that yields the negation of a boolean expression.

We now revise the definition of the transformation of phase 2 in light of the above discussion.
Definition 24 (Phase 2 - preliminary definition 2) The transformation
\( ph2'' \in SM \rightarrow SM \) which yields the inversion of well formed state machines is defined by
\[
ph2''((E, Q, R, q_1, F)) = (E, P(Q), R', \{q_1\}, P(Q))
\]
where the transition relation \( R' \) is defined by the following two rules:
\[
q_i \xrightarrow{(e,e,sa)} q' \in R \Leftrightarrow \{q_i\} \xrightarrow{(e,e,sa)} \{q'\} \in R'
\]

and
\[
\begin{align*}
\text{let } & \quad V_i(Q, e) \quad \text{def} \quad \{q \xrightarrow{(e',bx',as')} q' \in R \mid q \in Q \land e = e'\} \\
\text{let } & \quad V_i(Q, e, Ix) \quad \text{def} \quad \text{set}(Ix \uplus \text{list}(V_i(Q, e))) \\
\text{let } & \quad St(Q, e, Ix) \quad \text{def} \quad \{q \in Q \mid \exists q' \xrightarrow{(e',bx',as')} q'' \in V_i(Q, e, Ix) : q = q''\} \\
\text{let } & \quad Ga(Q, e, Ix) \quad \text{def} \quad \text{list}\{\{bx \in BExp | \\
& \quad \quad \quad \exists q' \xrightarrow{(e',bx',as')} q'' \in V_i(Q, e, Ix) : bx = bx'\}\} \\
\text{let } & \quad Ga'(Q, e, Ix) \quad \text{def} \quad \text{conj}(\{\text{conj}(Ga(Q, e, Ix), \\
& \quad \quad \quad \text{neg}(\text{disj}(Ga(Q, e, N \setminus Ix))))\}) \\
\text{let } & \quad As(Q, e, Ix) \quad \text{def} \quad \{as \in (\text{Var} \times \text{Exp})^* | \\
& \quad \quad \quad \exists q' \xrightarrow{(e',bx',as')} q'' \in V_i(Q, e, Ix) : as = as'\} \\
\text{let } & \quad As'(Q, e, Ix) \quad \text{def} \quad \text{flatten}\{\text{list}(As(Q, e, Ix))\} \\
\end{align*}
\]

\( \forall Q \in P(Q) : \\
\forall e \in E : \\
\forall Ix \in P(N) : \\
St(Q, e, Ix) \cap F = \emptyset \Leftrightarrow \\
Q \xrightarrow{(e,Ga'(Q,e,Ix),As'(Q,e,Ix))} (Q \cup St(Q, e, Ix)) \in R'
\]

The transformation \( ph2'' \) does not always yield the correct inversion of a state machine. For instance, consider the state machine \( W \) depicted on the left hand side of Fig. 16. It describes two traces: one trace consisting of 9 occurrences of \( a \), and one trace consisting of 9 occurrences of \( b \). Applying the phase 2 transformation \( ph2'' \) to \( W \) yields the state machine \( W' \) depicted on the right hand side of Fig. 16. Note that we have not depicted final states (since all states are final) or transitions whose guards always evaluate to false and that redundancy in the boolean expressions of the guards have been removed, e.g., \( i \leq 10 \) and \( \text{true} \) is written as \( i \leq 10 \).

The state machine \( W' \) rejects traces consisting of 10 or more occurrences of \( a \) or \( b \). For instance, the trace \( t \) consisting of 5 occurrences of \( a \) and 5 occurrences of \( b \) is rejected by the state machine \( W' \). However, no trace of \( W \) is a sub-trace of \( t \). Hence, \( W' \) is not a correct inversion of \( W \). The reason for this is that the two possible executions of \( W \), resulting from the branch in state 2, share the variable \( i \), i.e., the variable is used in a condition/guard of one execution and assigned to a value in another execution. In the example, this causes \( W' \) not to be a correct inversion of \( W \).

In general, to ensure that \( ph2'' \) yields the correction inversion \( SM' \) of a state machine \( SM \), we must require that all guards encountered when executing \( SM' \) must evaluate to the same values as the "corresponding" guards encountered when executing \( SM \). We say that a transformation is \textit{side effect free} for \( SM \) if this requirement is satisfied. This is precisely captured by the following definition.
Figure 16: State machine $W$ and its inversion

Figure 17: State machine $W$, its incorrect inversion $ph2''(W) = W'$, and its correct inversion $W''$

**Definition 25 (Side effect free)** Let $SM = (E, Q, R, q_I, F)$, $tr \in SM \to SM$, and $tr(SM) = SM' = (E', Q', R', q_I', F')$. Then transformation $tr$ is side
effect free for SM iff
\[
\forall s, t \in (\mathcal{E} \cup \{\epsilon\})^*: \forall (e, bx, as) \in \text{Act}_E:
\forall q_1, q_2 \in Q: \forall q'_1 \in Q' : \\
\forall \sigma_1, \sigma_1', \sigma_1'' \in \Sigma_T : \\
\wedge [q_1, \sigma_1] \xrightarrow{(e, bx, as)} [q_1, \sigma_1] \in E(G(SM)) \\
\wedge [q'_1, \sigma_1'] \xrightarrow{(e, bx, as)} [q'_1, \sigma_1'] \in E(G(SM')) \\
\wedge q_1 \xrightarrow{(e, bx, as)} q_2 \in R \\
\wedge \text{comp}[\langle SM \rangle, s, t] \\
\implies \sigma_1 \cap (\text{var}(bx) \times \text{Exp}) = \sigma_1' \cap (\text{var}(bx) \times \text{Exp})
\]

where the predicate \(\text{comp}(\_ , \_ ) \in \mathbb{P}(E^*) \times E^* \times E^* \rightarrow \mathbb{B}\) is defined \(\text{comp}(T, s, t) = \forall s \prec t \land \text{pr}(T, s) \neq \emptyset \land \neg (\exists s' \in \text{pr}(T, s) : s \sqsubseteq s' \land s' \prec t)\) where the function \(\text{pr} \in \mathbb{P}(E^*) \times E^* \rightarrow \mathbb{P}(E^*)\) is defined by \(\text{pr}(T, s) = \{s \sim s' \in T | (\exists t \in T : t \sqsubseteq s \sim s')\}\).

**Lemma 3** Let SM be a well formed state machine such that \(\emptyset \notin \sem{SM}\) and \(\text{ph}2''\) be side effect free for SM, then \(\text{ph}2''(SM)\) is an inversion of SM, i.e., \(\text{inv}(SM, \text{ph}2''(SM))\).

It is possible to define an alternative version of the transformation of phase 2 for which the side effect free condition is less restrictive. In particular, we observe that \(\text{ph}2''\) may generate unnecessary guards and assignment sequences for transitions corresponding to inconclusive behavior. As an example, consider the state machine \(W\) in Fig.17. It describes the trace \(\langle a, b \rangle\). The result of applying transformation \(\text{ph}2''\) to \(W\) is depicted by state machine \(W'\) in Fig.17 (i.e., \(\text{ph}2''(W) = W'\)). Note that we have not illustrated final states (since all states are final) or transitions whose guards always evaluate to false, and that boolean expressions have been simplified. The state machine \(W'\) is not a correct inversion of \(W\) since \(b\) is enabled after \(a\) has occurred 10 times. The problem is that the reflexive transition
\[
\{2, 3\} \xrightarrow{(a/i = i + 1)} \{2, 3\}
\]
in \(W'\) describing inconclusive behavior, contains the (unnecessary) assignment \(i = i + 1\) since \(W\) has the transition
\[
2 \xrightarrow{(a/i = i + 1)} 3
\]
which has been previously visited to reach the state \(\{2, 3\}\).

A solution to the problem of the current example, is the let each state of the inverted state machine record all transitions that are previously visited in order to reach that state. The previously visited transitions can then be disregarded when generating reflexive transitions corresponding to inconclusive behavior.

State machine \(W'\) in Fig.17 shows how this would work in the current example. Here \(V_1, V_2, V_3\) are sets of previously visited transitions of \(W\) defined by
\[
V_1 = \emptyset \quad V_2 = \{1 \xrightarrow{(i = 1)} 2\} \quad V_3 = \{1 \xrightarrow{(i = 1)} 2, 2 \xrightarrow{(a/i = i + 1)} 3\}
\]
Figure 18: State machine $W$ and its (incorrect) inversion $W'$ and (correct) inversion $W''$

Now, when generating transitions for $a$ in state $\{(2, 3), V_3\}$, we disregard the set $V_3$ of previously visited transitions. Thus we get

\[
\{(2, 3), V_3\} \xrightarrow{a} \{(2, 3), V_3\}
\]

and by definition of inversion (Def. 22) we have that $W''$ is a correct inversion of $W$.

The solution proposed above may not work for state machines that contain loops. For instance, consider the state machine $W$ of Fig.18. It describes the trace containing 9 occurrences of $a$ followed by $b$. In other words, the policy states that $b$ is not allowed to occur after $a$ has occurred 9 times. If we use the transformation $ph2''$ to invert $W$ and record previously visited transitions as described above, we get state machine $W'$ of Fig.18. Here we have that

\[
V_1 = \emptyset, \quad V_2 = \{1 \xrightarrow{i=0} 2\}, \quad V_3 = \{1 \xrightarrow{i=0} 2, 2 \xrightarrow{a[i < 10]/i=i+1} 2\}
\]

The state machine $W'$ is not a correct inversion of $W$ since it allows the occurrence of $b$ after $a$ has occurred more than 9 times. In this case, adding the transition $\{1 \xrightarrow{a[i < 10]/i=i+1} 2\}$ into the set of previously visited transitions, and thereby disregarding its transitions, is incorrect, because the transition is in a loop and may therefore be visited several times.

A solution to the problem is to remove the transitions of a loop from the set of visited transitions each time the loop is iterated. To achieve this, we can remove the outgoing transitions of each state that is entered from the set of previously visited transitions. In the current example, we would then obtain the state machine $W''$ of Fig.18 which is a correct inversion of $W$.

We are now ready to give the final definition of the transformation of phase 2.

**Definition 26 (Phase 2)** The transformation $ph2 : SM \rightarrow SM$ which yields the inversion of well formed state machines is defined by

\[
ph2((E, Q, R, q_I, F)) \xrightarrow{\text{def}} (E, (P(Q) \times P(R)), R', (\{q_I\}, \emptyset), (P(Q) \times P(R)))
\]
where the transition relation $R'$ is defined by the following two rules:

\[ q_i \xrightarrow{(e,sa)} q' \in R \iff (\{q_i\}, \emptyset) \xrightarrow{(e,sa)} (\{q'\}, \emptyset) \in R' \]

and

\[
\begin{align*}
\text{let } & Vi(Q) \equiv \{ q \mid q \in Q \} \\
Vi(Q,e,V) \equiv & \{ q \mid q \in Q \land e = e' \} \setminus V \\
Vi(Q,Ix,V) \equiv & \text{set}(Ix \oplus \text{list}(Vi(Q,e,V)) \\
St(Q,e,Ix,V) \equiv & \{ q \mid q \in Q \} \\
Ga(Q,e,Ix,V) \equiv & \text{list}(\{ bx \in \text{BExp} \mid \exists q' \mid (e',bx',as') \mid q'' \in Vi(Q,e,Ix,V) : q = q'' \} \\
Ga'(Q,e,Ix,V) \equiv & \text{conj}(\text{conj}(Ga(Q,e,Ix,V), \neg \text{disj}(Ga(Q,e,N \setminus Ix,V)))) \\
As(Q,e,Ix,V) \equiv & \{ as \in (\text{Var} \times \text{Exp})^* \mid \exists q' \mid (e',bx',as') \mid q'' \in Vi(Q,e,Ix,V) : as = as' \} \\
As'(Q,e,Ix,V) \equiv & \text{flatten}(\text{list}(As(Q,e,Ix,V)) \\
\text{in } & \forall (Q,V) \in \mathbb{P}(Q) \times \mathbb{P}(R) : \forall e \in \mathcal{E} : \forall Ix \in \mathbb{P}(N) : \\
& St(Q,e,Ix,V) \cap \mathcal{F} = \emptyset \iff \\
& (Q,V) \xrightarrow{(e, Ga'(Q,e,Ix,V), As'(Q,e,Ix,V))} \\
& (Q \cup St(Q,e,Ix,V), (V \cup Vi(Q,e,Ix,V) \setminus Vi(St(Q,e,Ix,V))) \in R' \\
\end{align*}
\]

**Corollary 1** Let $SM$ be a well formed state machine such that $() \not\in \llbracket SM \rrbracket$ and $ph2$ be side effect free for $SM$, then $ph2(SM)$ is an inversion of $SM$, i.e.,

\[ \text{inv}(SM, ph2(SM)) \]

The transformation $ph2$ will correctly invert the state machine examples of Fig. 14, Fig. 15, Fig. 17, and Fig. 18, as well as all the examples of the first part of this report (the part before the appendices).

However, $ph2$ does not work for the state machine of Fig. 16, where the variable $i$ is shared in the sense that it is used in a condition/guard of one execution and assigned to a value in another execution. We make this precise in the following definition.

**Definition 27 (Shared variables)** Let $SM = (\mathcal{E}, Q, R, q_1, \mathcal{F})$, then $SM$ does not have any shared variables iff

\[
\forall s,t,t' \in \text{Act}^*: q,q', q'' \in Q: \\
\forall(e_1,bx_1,as_1), (e'_1,bx'_1,as'_1), (e_2,bx_2,as_2), (e'_2,bx'_2,as'_2) \in \text{Act}: \\
\land q_1 \xrightarrow{r((e_1,bx_1,as_1) \text{ or } (e_2,bx_2,as_2))} q \in R \\
\land q_1 \xrightarrow{r((e'_1,bx'_1,as'_1) \text{ or } (e'_2,bx'_2,as'_2))} q' \in R \\
\land (e_1,bx_1,as_1) \neq (e'_1,bx'_1,as'_1) \\
\land \neg \forall \sigma \in \hat{\Sigma}_T : \text{eval}(\sigma(bx_1)) = \text{eval}(\sigma(bx'_1)) \\
\implies ((\text{var}(bx_2) \cap \text{avar}(as'_2)) \setminus \text{PVar}) = \emptyset
\]
where \( avar \in (\text{Var} \times \text{Exp})^* \rightarrow \mathbb{P}(\text{Var}) \) yields all the variables that are assigned to a value in an assignment sequence, i.e.,
\[
avar(((x_1, ex_1), \ldots, (x_n, ex_n))) = \{x_1\} \cup \cdots \cup \{x_n\}.
\]
We conjecture that if a state machine \( SM \) does not have shared variables in the sense of (Def. 27), then \( ph2 \) is side effect free for \( SM \) (Def. 25). By Corollary 1, this means \( ph2 \) will yield the correct inversion of any state machine that does not have shared variables.

In practice, the condition that a state machine policy must not have shared variables, means that when we compose several policies, then these policies cannot have the same variable names. For instance, consider again the state machine \( W \) of Fig. 16. This state machine may be seen as the composition of the two policies: (1) more than 9 occurrences of \( a \) is not allowed, and (2) more than 9 occurrences of \( b \) is not allowed. However, since both policies use the variable \( i \) to count the number of occurrences of \( a \) or \( b \), the condition of no shared variables is violated.

Note that it is feasible to automatically check whether a state machine has no shared variables because the condition is formulated in terms of the transitions of a state machine as opposed to the transitions of the execution graph.

Together, the transformation of phase 1 and phase 2 is adherence preserving when the condition of phase 2 is satisfied.

**Theorem 1** Let \( d \) be a well formed single lifeline sequence diagram such that \( ph2 \) is side effect free for \( ph1(\pi_l(d)) \) for all lifelines \( l \) in \( d \), then the transformation \( d2p(d) \) is adherence preserving, i.e.,
\[
d \rightarrow_{da} \Phi \Leftrightarrow d2p(d) \rightarrow_{sa} \Phi \text{ for all systems } \Phi
\]

### D.2 From general sequence diagrams to state machines

In this section, we define the transformation that takes a (general) sequence diagram and yields a set of state machines.

**Definition 28 (From sequence diagrams to sets of state machines)** The transformation \( d2pc \in D \rightarrow \mathbb{P}(SM) \) which takes a sequence diagram and yields a set of state machine, is defined by
\[
d2pc(d) \equiv \bigcup_{l \in \llbracket d \rrbracket} \{d2p(\pi_l(d))\}
\]

The transformation from sequence diagrams to state machine sets is adherence preserving when the condition of phase 2 is satisfied.

**Theorem 2** Let \( d \) be a well formed sequence diagram such that \( ph2 \) is side effect free for \( ph1(\pi_l(d)) \) for all lifelines \( l \) in \( d \), then the transformation \( d2pc(d) \) is adherence preserving, i.e.,
\[
d \rightarrow_{dag} \Phi \Leftrightarrow d2pc(d) \rightarrow_{sag} \Phi \text{ for all systems } \Phi
\]

### E Proofs

**Lemma 1** Let \( d \) be a well formed sequence diagram, then the state machine \( ph1(d) \) describes the negative traces of \( d \), i.e.,
\[
\llbracket ph1(d) \rrbracket = H_{neg} \quad \text{for} \quad \llbracket d \rrbracket = (H_{pos}, H_{neg})
\]
Proof of Lemma 1  By Lemma 1.1 and Lemma 1.2, and definition of $ph_1$ (Def. 21).

Lemma 1.1  Let $d$ be a well formed sequence diagram, $SM = (eca.d \cap E, Q, R, q_I, F)$ and $SM' = (\psi(\mathcal{E}), Q, \psi(R), q_I, F)$ for some variable renaming function $\psi \in \text{PVar} \rightarrow \text{PVar}$ satisfying constraint (26). Then the semantics of $SM$ is equal to the semantics of $SM'$, i.e.,

\[
\llbracket SM \rrbracket = \llbracket SM' \rrbracket
\]

Proof of Lemma 1.1  By definition of the execution graph of state machines (Def. 11), all parameter variables are treated as local variables for each transition, thus a renaming of the parameter variables has no effect on the execution unless two distinct parameter variables of the same signal are renamed into the same parameter variable. However, this cannot occur since $\psi$ is assumed to satisfy constraint (26). Note that it is always possible to find a renaming function that satisfies constraint (26) for the events of a given well formed sequence diagram because well formed diagrams must satisfy conditions SD3 and SD7.

Lemma 1.2  Let $d$ be a well formed single lifeline sequence diagram such that $\llbracket d \rrbracket = (H_{\text{pos}}, H_{\text{neg}})$, let $ph_1(d) = (\psi(\mathcal{E}), Q, \psi(R), q_I, F)$ for some parameter variable renaming function $\psi \in \text{PVar} \rightarrow \text{PVar}$, and let $SM = (E, Q, R, q_I, F)$, then the semantics of $SM$ is equal to $H_{\text{neg}}$, i.e.,

\[
\llbracket SM \rrbracket = H_{\text{neg}}
\]

Proof of Lemma 1.2

Assume:
1. $ph_1(d) = (\psi(\mathcal{E}), Q, \psi(R), q_I, F)$ and $SM = (E, Q, R, q_I, F)$ for some $d \in \mathcal{D}^l$, $l \in \mathcal{L}$, and $\psi \in \text{PVar} \rightarrow \text{PVar}$
2. $\llbracket d \rrbracket = (H_{\text{pos}}, H_{\text{neg}})$
3. $d$ is well formed (i.e., $d$ satisfies conditions SD1 - SD10)

Prove: $\llbracket SM \rrbracket = H_{\text{neg}}$

(1) Assume: 1.1. $s \in H_{\text{neg}}$

Prove: $s \in \llbracket SM \rrbracket$

(2)1. $\exists s' \in (\hat{E} \cup \{\epsilon\})^*, \sigma_I, \sigma \in \hat{\Sigma}_T, q \in F$:

$\land [q_I, \sigma_I] \xrightarrow{s'} [q, \sigma] \in \text{EG}(SM)$

$\land s'|_{\hat{E}} = s$

(3)1. Choose $(e_1, \ldots, e_n) \in \hat{E}^*$ such that $(e_1, \ldots, e_n) = s$

Proof: By assumption 2, assumption 1.1, and definition of $\llbracket - \rrbracket$ (Def. 7).

(3)2. Choose

$t \in (\hat{E} \cup \{t, f, \bot, \tau_{\text{alt}}, \tau_{\text{refuse}}, \tau_{\text{loop}}\})^*$, $\sigma_I, \sigma_n \in \hat{\Sigma}_T$, and $\beta_n \in \mathcal{B}$

such that

$[\llbracket [0, l.l.d], d, \sigma_I \rrbracket \xrightarrow{t} [\beta_n, \text{skip}, \sigma_n]]$

$t|_{\tau_{\text{refuse}}, t, \bot} \in \{\tau_{\text{refuse}}\}^+$

$t|_{\hat{E}} = s$

Proof: By assumption 2, assumption 1.1, and definition of $\llbracket - \rrbracket$ (Def. 7).
(3.3) Choose
\[ tt_1, \ldots, tt_n \in \{t\}^*, \]
\[ tt_0, tt_1, \ldots, tt_n \in \{\tau_{assign}\}^* \text{, and} \]
\[ E = (E \cup \{t, \tau_{assign}\}) \setminus \{\tau_{alt}, \tau_{loop}, \tau_{refuse}\} \text{ such that} \]
\[ t|E = tt_0 \wedge tt_1 \wedge (e_1) \wedge tt_1 \wedge \cdots \wedge tt_n \wedge (e_n) \wedge tt_n \]

**Proof:** By (3.1) and (3.2) we know that \( t|E = \langle e_1, \ldots, e_n \rangle \). Furthermore, since \( d \) satisfies syntax constraints \( SD4 \) and \( SD8 \) (by assumption 3), we know that \( t \) must be of the form asserted by (3.3) by definition of the execution system for sequence diagrams (Def. 6).

(3.4) \[ \left\langle (0, ll.d), d, \sigma_1 \right\rangle \xrightarrow{tt_0=tt_{1-1}(e_1) \cdots tt_n=tt_{n-1}(e_n)} [\beta_n, \text{skip}, \sigma_n] \]

**Proof:** By (3.2), (3.3) and definition of \( \Rightarrow \) (Def. 19).

(3.5) Choose
\[ \sigma_0, \ldots, \sigma_{n-1} \in \Sigma, \]
\[ \beta_0, \beta_1, \ldots, \beta_{n-1} \in B, \]
\[ d_0, \ldots, d_{n-1} \in D \]

such that
\[ \left\langle (0, ll.d), d, \sigma_1 \right\rangle \xrightarrow{tt_0} [\beta_0, d_0, \sigma_0] \xrightarrow{tt_{1-1}(e_1)} [\beta_1, d_1, \sigma_1] \cdots \xrightarrow{tt_{n-1}(e_n)} [\beta_n, \text{skip}, \sigma_n] \]

**Proof:** By (3.4).

(3.6) Choose
\[ tc_1, \ldots, tc_n \in C^*, \]
\[ ta_0, \ldots, ta_n \in A^*, \text{ and} \]
\[ e'_1, \ldots, e'_n \in E \]

such that
\[ \Pi'(ll.d, d) \xrightarrow{ta_0} \Pi'(ll.d, d_0) \xrightarrow{tt_{1-1}(e'_1)} \Pi'(ll.d, d_1) \cdots \xrightarrow{tt_{n-1}(e'_n)} \Pi'(ll.d, \text{skip}), \]
\[ eval(\sigma_i(e'_i)) = e_i \text{ for all } i \in \{1, \ldots, n\}, \]
\[ eval(\sigma_i(c2g(tc_i)) = t \text{ for all } i \in \{1, \ldots, n\}, \]
\[ \sigma_0 = as2ds(\sigma_1, a2ef(ta_0)), \text{ and} \]
\[ \sigma_{i+1} = as2ds(\sigma_i, a2ef(ta_i)) \text{ for all } i \in \{0, \ldots, n-1\} \]

**Proof:** By Def.6, a transition \( \Pi'(ll.d, d') \xrightarrow{cca} \Pi'(ll.d, d'') \) of the revised projection system corresponds to the transition \( [\beta', d', \sigma'] \xrightarrow{eval(cca)} [\beta'', d'', \sigma'] \) of the execution system if \( cca \) is an event or constraint, or the transition \( [\beta', d', \sigma'] \xrightarrow{\tau_{assign}} [\beta'', d'', \sigma''] \) where \( \sigma'' = \sigma[x \mapsto eval(ex)] = as2ds(\sigma'(c, cca)) \) if \( cca \) is an assignment of the form \( (x, c) \). By (3.5), definition of the execution system for sequence diagrams (Def.6), definition of \( c2g \) (Eq. (25)), \( a2ef \) (Eq. (25)), and \( as2ds \) (Eq. (24)), we therefore have that (3.6) holds.

(3.7) Let: Let \( pr \) be a function that sets the index of all parameter variables in a term to zero, e.g., \( pr((v, n, 3)) = (v, n, 0) \)

(3.8) \[ \Pi(ll.d, d) \xrightarrow{ta_0} \Pi(ll.d, d_0) \xrightarrow{pr(tc_1) \cdots pr(tc_n)} \Pi(ll.d, pr(d_1)) \]
\[ \cdots \xrightarrow{pr(tc_n) \cdots pr(tc_n)} \Pi(ll.d, pr(d_n)) \]

**Proof:** By (3.6), (3.7), and definition of the projection systems for sequence diagrams (Def.4 and Def. 5).

(3.9) Choose
\[ (Q_0, mo_0), \ldots, (Q_n, mo_n) \in Q \]
such that
\[
\begin{align*}
&([d], pos) \xrightarrow{(e, \text{a2ef}(ta_0))} (Q_0, mo_0) \xrightarrow{(pr(e'_1), pr(c2g(tc_1)), pr(a2ef(ta_1)))} \\
&(Q_1, mo_1) \cdots \xrightarrow{(pr(e'_n), pr(c2g(tc_n)), pr(a2ef(ta_n)))} (Q_n, mo_n) \in \mathcal{R},
\end{align*}
\]

pr(d_i) \in Q_i for all i \in \{0, \ldots, n - 1\}, and skip \in Q_n

PROOF: By definition of ph1 (Def 21), a transition \( \Pi(l.l.d, d') \xrightarrow{\text{tc}(e)-\text{ta}} \tau_{mo'} \)
\( \Pi(l.l.d, d'') \) of the projection system corresponds to a transition
\( (Q', mo) \xrightarrow{c2g(tc)\text{a2ef}(ta)} (Q'', mo') \) of the state machine (which is produced by ph1) where \( d' \in Q' \) and \( d'' \in Q'' \). Therefore, by assumption \( 1 \) (since ph1(\( d = SM \)), (3)8, and definition of ph1 (Def 21), we have that (3)9 holds.

(3)10. \((Q_n, mo_n) \in \mathcal{F}\)

PROOF: By definition of ph1 (Def 21), \((Q_n, mo_n) \) is in the set of final states \( \mathcal{F} \) if skip \( \in Q_n \) and \( mo_n = \text{neg} \). By (3)9, we know that \( \text{skip} \in Q_n \).

To see that \( mo_n = \text{neg} \), note that by (3)2, \( t \) must contain the silent event \( \tau_{\text{refuse}} \). This means that \( ([0, l.l.d, d, \sigma_I] \xrightarrow{\text{-}} \tau_{\text{neg}} [\beta_f, \text{skip}, \sigma_\text{n}] \) holds by definition of \( \tau_{\text{refuse}} \) (Def 20). By (3)5 - (3)9 and definition of ph1 (Def 21), this implies that \( mo_n = \text{neg} \).

(3)11. \( \exists \sigma'_0, \ldots, \sigma'_n \in \hat{\Sigma}_T: \)
\( \left[ ([d], pos), \sigma_1 \right] \xrightarrow{\Delta} \left[ (Q_0, mo_0), \sigma'_0 \right] \xrightarrow{\sigma_1} \left[ (Q_1, mo_1), \sigma'_1 \right] \cdots \)
\( \xrightarrow{\sigma_n} \left[ (Q_n, mo_n), \sigma'_n \right] \in \mathcal{E}(SM) \)

(4)1. Choose
\( \sigma'_0, \ldots, \sigma'_n \in \hat{\Sigma}_T \) and
\( \sigma''_0, \ldots, \sigma''_n \in \hat{\Sigma} \)
such that
\( (A) \text{Dom}(\sigma''_i) = \text{pvar}(pr(e'_i)) \) for all \( i \in \{1, \ldots, n\} \),
\( (B) \text{eval}(\sigma'_i[\sigma''_i](pr(e'_i))) = e_i \) for all \( i \in \{1, \ldots, n\} \),
\( (C) \sigma''_0 = \text{as2ds}(\tau_{\text{neg}}) \), and
\( (D) \sigma_{i+1}' = \text{as2ds}(\sigma''_i[pr(a2ef(ta_i))]) \) for all \( i \in \{0, \ldots, n - 1\} \)

PROOF: Data states that satisfy (A), (C), and (D) of (4)1 can always be chosen trivially. Furthermore, data states that satisfy (B) can be chosen because the message of each event \( pr(e'_i) \) contain parameter variables only since \( d \) is assumed to satisfy syntax constraint SD3 by assumption 3. Therefore it is always possible to chose a data state \( \sigma''_i \) such that \( \text{eval}(\sigma'_i[\sigma''_i](pr(e'_i))) = \text{eval}(\sigma''(pr(e'_i))) = \sigma''_i(pr(e'_i)) = e_i \).

(4)2. \( \text{eval}(\sigma'_i[\sigma''_i](pr(c2g(tc_i)))) = t \) for all \( i \in \{1, \ldots, n\} \)

PROOF: By (3)6, (3)7, (3)8, and (4)1.

(4)3. Q.E.D.

PROOF: By (4)1, (4)2, and definition of the execution graph for state machines (Def. 11).

(3)12. Q.E.D.

PROOF: By (3)10 and (3)11.

(2)2. Q.E.D.

PROOF: By (2)1 and definition of \( \cdot \) (Def.12).

(1)2. ASSUME: 1.1. \( s \in \left[ \left[ SM \right] \right] \)

PROVE: \( s \in H_{\text{neg}} \)

(2)1. \( \exists \sigma, \sigma' \in \hat{\Sigma}_T, \beta \in B, s' \in (\mathcal{E} \cup \{ t, f, \perp, \tau_{\text{alt}}, \tau_{\text{refuse}}, \tau_{\text{loop}} \}^* : \)
E PROOFS

\[ \forall \langle d, \sigma \rangle \in (\mathbb{E}, \mathbb{D}, \mathbb{D}, \mathbb{D}) \cdot \exists \sigma' \cdot \exists \beta, \text{skip} \cdot \exists \bar{s} \cdot \bar{s}'(\beta, \text{skip}, \sigma') \]

\( \wedge \bar{s}'[\mathbb{E}] = s \)

\( \wedge \bar{s}'[\{t, \tau_{\text{reset}}\}] = \{\tau_{\text{reset}}\}^+ \)

(3.1) Choose 

\( t \in (\mathbb{E} \cup \{\epsilon\})^* \), 

\( \sigma_1, \sigma_n \in \sum_T \), 

\( (Q_n, m_0) \in \mathbb{F} \) 

such that 

\[ [q_t, \sigma_i] \bar{t} \{(Q_0, \text{neg}), \sigma_0\} \]

\[ \bar{t} \{(Q_1, \sigma_1)\} \]

\[ \bar{t} \{(Q_n, \text{neg}), \sigma_n\} \in \mathbb{E}(SM) \]

\( s = t \bar{t} \)

PROOF: By assumption 1, assumption 1.1, definition of \( \lceil \rceil \) (Def. 12), and definition of \( \text{ph1} \) (Def. 21).

(3.2) \( t = (\epsilon) \bar{t} \)

PROOF: By definition of \( \text{ph1} \) (Def. 21), \( SM \) has exactly one transition from its initial state, and that transition is not labeled by an action containing an event. All other transitions of \( SM \) are labeled by actions containing events. By definition of \( \lceil \rceil \) (Def. 12), this means that \( t = (\epsilon) \bar{t} \).

(3.3) Choose \( (e_1, \ldots, e_n) \in \mathbb{E}^* \) such that \( s = (e_1, \ldots, e_n) \)

PROOF: By (3.1) and definition of the execution graph of state machines (Def. 11).

(3.4) Choose 

\[ (Q_0, m_0), (Q_1, m_0), \ldots, (Q_{n-1}, m_0) \in \mathbb{Q} \]

\( \sigma_0, \sigma_1, \ldots, \sigma_{n-1} \in \sum_T \)

such that 

\[ [q_t, \sigma_0] \bar{t} \{(Q_0, m_0), \sigma_0\} \]

\[ \bar{t} \{(Q_1, \sigma_1)\} \]

\[ \bar{t} \{(Q_n, \text{neg}), \sigma_n\} \in \mathbb{E}(SM) \]

PROOF: By (3.1), (3.2), (3.3), and definition of the execution graph of state machines (Def. 11).

(3.5) Choose 

\( e_1', \ldots, e_n' \in \mathbb{E}, \)

\( b x_1, \ldots, b x_n \in \mathbb{BExp}, \)

\( a s_0, \ldots, a s_n \in (\mathbb{V} \times \mathbb{E})^*, \) and 

\( \sigma_1', \ldots, \sigma_n' \in \sum_T \)

such that 

\[ q_1 \circ (e_1', a s_0) \rightarrow (Q_0, m_0) \]

\[ \rightarrow (Q_1, \sigma_1'), (Q_2, \sigma_2), \ldots \]

\[ \rightarrow (Q_n, \text{neg}) \in \mathbb{R}, \]

\( \text{Dom}(\sigma_i') = \text{var}(e_i') \) for all \( i \in \{1, \ldots, n\}, \)

\( \sigma_0 = \text{as2ds}(a s_0), \)

\( \sigma_{i+1} = \text{as2ds}(\sigma_i, a s_i) \) for all \( i \in \{0, \ldots, n-1\}, \)

\( \text{eval}(\sigma_i', e_i') = e_i \) for all \( i \in \{0, \ldots, n\}, \) and 

\( \text{eval}(\sigma_i', (b x_i)) = t \) for all \( i \in \{0, \ldots, n\} \)

PROOF: By assumption 1, \( \mathbb{R} \) denotes the transitions of \( SM \), therefore, (3.5) holds by (3.4) and definition of the execution graph for state machines (Def. 11).

(3.6) Choose 

\( d_0 \in Q_0, \ldots, d_{n-1} \in Q_{n-1}, \)

\( t c_1, \ldots, t c_n \in \mathbb{C}^*, \) and 

\( t a_0, t a_1, \ldots, t a_n \in \mathbb{A}^* \)
such that
\[
(A) \text{ } \Pi(\langle l.l, d \rangle) \xrightarrow{\text{tn}} \Pi(\langle l.l, d_0 \rangle) \xrightarrow{\text{tc} - \langle e_i' \rangle \rightarrow \text{ta}_i} \Pi(\langle l.l, d_1 \rangle) \ldots
\]
\[
\xrightarrow{\text{tc} - \langle e_i' \rangle \rightarrow \text{ta}_i} \Pi(\langle l.l, \text{skip} \rangle),
\]
\[(B) \text{ } bx_i = c2g(tc_i) \text{ for all } i \in \{1, \ldots, n\},
\]
\[(C) \text{ } as_i = a2ef(ta_i) \text{ for all } i \in \{0, \ldots, n\}, \text{ and } (D)
\]
\[\Pi(\langle l.l, d_j \rangle) \xrightarrow{\text{tc} - \langle e_i'' \rangle \rightarrow \text{as}_j} \Pi(\langle l.l, d_{j+1} \rangle)
\]
for some \( j \in \{0, \ldots, n - 1\}
\]
or
\[
\Pi(\langle l.l, d \rangle) \xrightarrow{\text{tn}} \Pi(\langle l.l, d_0 \rangle)
\]

**Proof:** By definition of \( \text{ph1} \) (Def 21), a transition \( \Pi(\langle l.l, d' \rangle) \xrightarrow{\text{ec2g(tc), a2ef(ta)}} \Pi(\langle l.l, d'' \rangle) \) of the state machine (which is produced by \( \text{ph1} \)) where \( d' \in Q' \) and \( d'' \in Q'' \). Therefore, by assumption 1 (since \( \text{ph1}(d) = \text{SM} \)), (3)5, and definition of \( \text{ph1} \) (Def 21), we have that
(3)7. Let: Let \( pr \) be a function that sets the index of all parameter variables in a term to zero, e.g., \( pr((vn, 3)) = (vn, 0) \)

(3)8. Choose
\[
d_1', \ldots, d_n' \in D,
\]
\[
tc_1', \ldots, tc_n' \in C^*,
\]
\[
e_1'', \ldots, e_n'' \in E, \text{ and}
\]
\[
ta_1', \ldots, ta_n' \in A^*
\]
such that
\[
pr(d_i') = d_i \text{ for all } i \in \{1, \ldots, n\},
\]
\[
pr(tc_i') = tc_i \text{ for all } i \in \{1, \ldots, n\},
\]
\[
pr(e_i'') = e_i' \text{ for all } i \in \{1, \ldots, n\},
\]
\[
pr(ta_i') = ta_i \text{ for all } i \in \{1, \ldots, n\}, \text{ and}
\]
\[
\Pi'(\langle l.l, d \rangle) \xrightarrow{\text{tn}} \Pi'(\langle l.l, d_0 \rangle) \xrightarrow{\text{tc} - \langle e_i'' \rangle \rightarrow \text{ta}_i} \Pi'(\langle l.l, d_1' \rangle) \ldots
\]
\[
\xrightarrow{\text{tc} - \langle e_i'' \rangle \rightarrow \text{ta}_i} \Pi'(\langle l.l, d_n' \rangle)
\]

**Proof:** By (3)6, definition of \( pr \) ((3)7) and definition of the revised projection system (Def. 5).

(3)9. \( \exists \theta_1, \ldots, \theta_n \in \{t\}^* \): 
\[
\exists \theta_0, \ldots, \theta_n \in \{\text{assign}\}^* : 
\]
\[
\exists \beta_0, \ldots, \beta_n \in B : 
\]
\[
\exists \sigma_1, \sigma_0', \sigma_1', \ldots, \sigma_n' \in \Sigma : 
\]
\[
[(\emptyset, ll.d), d, \sigma_1'] \xrightarrow{\text{tn}} [\beta_0, d_0, \sigma_0'] \xrightarrow{\text{tn}} [\beta_1, d_1', \sigma_1'] \ldots
\]
\[
\xrightarrow{\text{tn}} [\beta_n, d_n', \sigma_n']
\]

(4)1. Choose 
\[
\sigma_1', \sigma_0', \ldots, \sigma_n' \in \Sigma
\]
such that
\[
(A) \text{ } \sigma_1' \setminus (\text{PVar} \times \text{Exp}) = \sigma_1 \setminus (\text{PVar} \times \text{Exp}),
\]
\[
(B) \text{ } \sigma_0' = as2ds(\sigma_1', a2ef(ta_0)),
\]
\[
(C) \text{ } \sigma_{i+1} = as2ds(\sigma_i', a2ef(ta_i')) \text{ for all } i \in \{0, \ldots, n - 1\}, \text{ and}
\]
\[
(D) \text{ } \text{eval}(\sigma_i'(e_i'')) = e_i \text{ for all } i \in \{1, \ldots, n\}
\]
Proof: Conditions (A), (B), and (C) of (4) hold trivially. Condition (D) holds because the signal of each event $e_i''$ contains parameter variables only and no two events contain the same parameter variables (since $d$ satisfies syntax constraints $SD_3$ and $SD_7$ by assumption 3). Therefore, we can always choose an initial state $\sigma_i'$ that assigns variables to values such that (D) holds since parameter variables are never explicitly assigned to values by any assignment sequence $ta_i'$ (since only a normal variable and not a parameter variable can be explicitly assigned to a value by the constructs in a sequence diagram).

(4.2) $eval(\sigma_i(c_2(q(t\sigma_i')))) = t$ for all $i \in \{1, \ldots, n\}$

Proof: By (3)4 and (4)1.

(4.3) Q.E.D.

Proof: By (4)1 and (4)2.

(3)10. Q.E.D.

Proof: By (3)9, we know that $[([(), l.l, d), d, \sigma_i'] \rightarrow [\beta_n, d_n', \sigma_n']$ for some trace $t$ such that $l(t) = s$. Furthermore, $d_n' = \text{skip}$ by (3)6 and (3)8, and $t$ must contain the silent event $\tau_{\text{refuse}}$ by (3)6. Therefore (2)1 must hold.

(2.2) Q.E.D.

Proof: By (2)1 and definition of $[\ ]$ (Def. 7).

(1)3. Q.E.D.

Proof: By (1)1 and (1)2.

Lemma 2 Let $SM$ be a state machine whose transitions each contain exactly one event (whose signal has zero arguments) and no guards or assignments. Then $ph2'(SM)$ is an inversion of $SM$ if $\langle \rangle \notin [SM]$, i.e.,

$$\langle \rangle \notin [SM] \implies inv(SM, ph2'(SM))$$

Proof of Lemma 2

Assume:

1. $SM = (E, Q, R, q_1, F)$ and $\langle \rangle \notin [SM]$
2. $q \xrightarrow{act} q' \in R \implies act = (e, e, e)$ for some $e \in E$ whose signal has zero arguments
3. $SM' = ph2'(SM) = (E', Q', R', Q_1, F')$
4. $s \in \{e \in \hat{E}| \exists e' \in E : e = e'\}^*$

Prove: $(\forall t \in [SM] : -(t \circ s)) \iff s \in [SM']$

(1) Assume: (1.1) $(\forall t \in [SM] : -(t \circ s))$

Prove: $s \in [SM']$

(2)1. $\forall sp \in \hat{E}^*: sp \subseteq s \implies sp \in [SM']$

(3)1. Assume: 2.1 $sp \subseteq s$ for some $sp \in \hat{E}^*$

Prove: $sp \in [SM']$

(4)1. Case: 3.1. $sp = \langle \rangle$

Proof: $Q_t \in F'$ by definition of $ph2'$ (Def. 23) and assumption 3. This means that $\langle \rangle \in [SM']$ by definition of $[\ ]$ (Def. 12).

(4)2. Case: 3.1. $sp = sp' \sim (e)$ for some $sp' \in [SM']$ and $e \in \hat{E}$

(5)1. Choose $e_1, e_2, \ldots, e_n \in E$ such that $sp' = \langle e_1, e_2, \ldots, e_n \rangle$

Proof: By assumption 2.1 and assumption 3.1.

(5)2. Choose $Q_1, Q_2, \ldots, Q_n \in Q'$ such that $Q_t \xrightarrow{act} Q_1 \xrightarrow{act} Q_2 \cdots \xrightarrow{act} Q_n \in R'$
PROOF: By assumption 2, case assumption 3.1, ⟨5⟩1, and definition of [] (Def.12).

⟨5⟩3. Choose \( Q_{n+1} \in \mathcal{F} \) such that \( Q_n \not\rightarrow Q_{n+1} \in \mathcal{R'} \)

⟨6⟩1. Assume: 4.1 \(~(\exists Q_{n+1} : Q_n \not\rightarrow Q_{n+1} \in \mathcal{R'})\)  

PROVE: False

⟨7⟩1. Choose \( q_n \in Q_n \) and \( q_f \in \mathcal{F} \) such that \( q_n \not\rightarrow q_f \) \( \in \mathcal{R} \)  

PROOF: Assumption 4.1 implies that \( St(Q_n, e) \cap \mathcal{F} = \emptyset \) where \( St \) is the function defined in (Def.23) (otherwise we would have \( Q_n \not\rightarrow Q_n \cup St(Q_n, e) \) which contradicts assumption 4.1). By ⟨5⟩2, assumption 3, and definition of \( ph2' \) (Def. 23), this means that ⟨7⟩1 holds.

⟨7⟩2. Choose \( u \in \mathcal{E}^* \) such that \( q_f \not\rightarrow q_n \) \( \in \mathcal{R} \) and \( u \prec sp' \)  

PROOF: By definition of \( ph2' \), we know that for each transition \( q' \not\rightarrow q'' \) (where \( q'' \notin \mathcal{F} \)) of \( SM \), there is a corresponding transition \( Q' \not\rightarrow Q'' \) (where \( q' \in Q' \) and \( q'' \in Q'' \)) in the inverted state machine \( SM' \). Since we have \( Q_f \not\rightarrow sp' \) \( Q_n \in \mathcal{R'} \) by ⟨5⟩1 and ⟨5⟩2, it is easy to see that we can choose \( u \) such that \( q_f \not\rightarrow q_n \) \( \in \mathcal{R} \) and \( u \prec sp' \).

⟨7⟩3. \( u \prec \langle e \rangle \in [SM] \)  

PROOF: By ⟨7⟩1, ⟨7⟩2, assumption 1, and definition of [] (Def.12).

⟨7⟩4. \( u \prec \langle e \rangle \prec sp' \prec \langle e \rangle \)  

PROOF: By ⟨7⟩2 and definition of \( \prec \) (Eq. (20)).

⟨7⟩5. Q.E.D.

PROOF: By assumption 1.1, no trace in [SM] can be a subtrace of \( s \). However, by ⟨7⟩3 \( u \prec \langle e \rangle \) is in [SM]. Furthermore, \( u \prec \langle e \rangle \) is a subtrace of \( s \) because \( u \prec \langle e \rangle \) is a subtrace of \( sp \) (by ⟨7⟩4 and assumption 3.1) which is a prefix of \( s \) (by assumption 2.1).

Hence assumption 4.1 cannot hold.

⟨6⟩2. Q.E.D.

PROOF: By contradiction.

⟨5⟩4. Q.E.D.

PROOF: By case assumption 3.1, ⟨5⟩1, ⟨5⟩2, ⟨5⟩3, assumption 3, and definition of [] (Def.12).

⟨4⟩3. Q.E.D.

PROOF: By ⟨4⟩1, ⟨4⟩2, and induction over the length of \( sp \).

⟨3⟩2. Q.E.D.

PROOF: By \( \forall \)-rule.

⟨2⟩2. Q.E.D.

PROOF: By ⟨2⟩1.

⟨1⟩2. Assume: 1.1 \( s \in [SM'] \)  

PROOF: \((\forall t \in [SM] : \neg(t \prec s))\)

⟨2⟩1. Assume: 2.1 \( t \prec s \) for some \( t \in [SM] \)  

PROOF: False

⟨3⟩1. Choose \( t' = \langle e_1, \ldots, e_n \rangle \in \mathcal{E}^* \) such that \( t' \subseteq t \), and choose \( q_1, \ldots, q_{n-1} \in Q \setminus \mathcal{F} \) and \( q_n \in \mathcal{F} \) such that \( q_t \not\rightarrow q_1 \not\rightarrow q_2 \cdots q_{n-1} \not\rightarrow q_n \) \( \in \mathcal{R} \)  

PROOF: By assumption 2, assumption 2.1, assumption 1 (since \( \langle \rangle \notin [SM] \)), and definition of [] (Def.12).

⟨3⟩2. Choose \( u \in \mathcal{E}^* \) such that \( u \prec \langle e_n \rangle \subseteq s \) and \( \langle e_1, e_2, \ldots, e_n \rangle \prec u \) and
Lemma 3 Let $SM$ be a well formed state machine such that $\langle \rangle \notin [SM]$ and $ph2''$ be side effect free for $SM$, then $ph2''(SM)$ is an inversion of $SM$, i.e.,

$$inv(SM, ph2''(SM))$$

Proof of Lemma 3

**Assume:**
1. $SM = (E, Q, R, q_I, F)$ for $SM \in SM$ satisfying SM1 - SM4 and $\langle \rangle \notin [SM]$
2. $ph2''$ is side effect free for $SM$
3. $SM' = ph2''(SM) = (E', Q', R', q_I', F')$
4. $s \in \{ e \in \tilde{E} | \exists e' \in \mathcal{E} : e = e' \}$

**Prove:** $(\forall t \in [SM] : \neg(t \circ s)) \Leftrightarrow s \in [SM']$

(1.1) **Assume:** $1.1. \ (\forall t \in [SM] : \neg(t \circ s))$

**Prove:** $s \in [SM']$

(2.1) **Case:** $3.1. \ sp = \langle \rangle$

*Proof:* $q_I' \in F'$ by definition of $ph2''$ (Def. 24) and assumption 3. This means that $\langle \rangle \in [SM']$ by definition of $[ ]$ (Def.12).

(4.1) **Case:** $3.1. \ sp = sp' \setminus \langle \rangle$ for some $sp' \in [SM']$ (the induction hypothesis) and $e \in \tilde{E}$

(5.1) **Choose:** $e_1, e_2, \ldots, e_n \in \tilde{E}$ such that $sp' = \langle e_1, e_2, \ldots, e_n \rangle$

*Proof:* By assumption 2.1 and assumption 3.1.

(5.2) **Choose:** $t \in (E \cup \{\langle \rangle\}^*, Q_n \in Q'$ and $\sigma_I, \sigma'_n \in \tilde{\Sigma}_T$ such that $[q_I', \sigma_I] \xrightarrow{\sigma_I} [Q_n, \sigma'_n] \in EG(SM')$ and $t|_E = sp'$

*Proof:* By case assumption 3.1 (since $sp' \in [SM']$) and definition of $[ ]$ (Def. 12).

(5.3) **Choose:** $t = \langle \rangle \setminus sp'$

*Proof:* By assumption 1, $SM$ is assumed to satisfy syntax constraints SM2 and SM3 that ensure that all transitions of $SM$ except the outgoing transition for the initial state of $SM$ are labeled
by actions containing events. By definition of \( ph2'' \) (Def. 24) this means that the same constraints hold for the inverted state machine \( SM' \). Therefore we can assert that \( t = \langle e \rangle \rightarrow sp' \).

(5.4) Choose
\[
Q_0, Q_1, Q_2, \ldots, Q_{n-1} \in Q' \text{ and } \\
\sigma_0, \ldots, \sigma_{n-1} \in \hat{\Sigma}_T
\]
such that
\[
[q_1, \sigma_1] \xrightarrow{\tau} [q_0, \sigma_0] \xrightarrow{\tau} [Q_1, \sigma_1] \xrightarrow{\tau} [Q_2, \sigma_1] \ldots \\
\xrightarrow{\tau} [Q_n, \sigma_n] \in EG(SM')
\]

PROOF: By (5.1), (5.2), and (5.3).

(5.5) Choose \( Q_{n+1} \in F' \) and \( \sigma'_{n+1} \in \hat{\Sigma}_T \) such that
\[
[q_n, \sigma_n] \xrightarrow{\tau} [Q_{n+1}, \sigma'_{n+1}] \in EG(SM')
\]

(6.1) Choose
\[
q_1, \ldots, q_k \in Q_n, \\
p_1, \ldots, p_k \in Q, \\
e' \in E, \\
bn_1, \ldots, bn_k \in BExp, \text{ and } \\
as_1, \ldots, as_k \in (Var \times Exp)^*
\]
such that
\[
((q_1, (e', bn_1, as_1), p_1), \ldots, (q_k, (e', bn_k, as_k), p_k)) = \\
list\{(q, (e'', bn, as), e') \in R \mid q \in Q_n \land e = e''\}
\]

PROOF: By assumption 1, definition of the alphabet of state machines (Def. 10), and definition of the list function (see App. D.1.2).

(6.2) Choose \( \phi \in \hat{\Sigma} \) such that \( \text{Dom}(\phi) = \text{pvar}(e') \) and \( \phi(e') = e \)

PROOF: By (6.1) and definition of the alphabet of state machines (Def. 10).

(6.3) Choose \( Ix \subseteq \{1, \ldots, n\} \) such that \( i \in Ix \) iff \( \text{eval}(\sigma'_{n}[\phi](bx_{i})) = \tau \)

PROOF: Trivial.

LEt: \( bx = (bx_1, bx_2, \ldots, bx_k) \)

LET: \( bx'' = \text{conj}(\text{conj}(Ix \oplus bx), \text{neg}(\text{disj}(\{1, \ldots, n\} \setminus Ix \oplus bx))) \)

(6.4) \( \text{eval}(\sigma'_{n}[\phi](bx'')) = \tau \)

PROOF: By (6.3), definition of \( bx'' \), and definition of \( \text{conj}, \text{disj}, \text{and neg} \) (Sect. D.1.2).

(6.5) \( \text{set}(Ix \oplus (p_1, \ldots, p_k)) \cap F = \emptyset \)

(7.1) Assume: 4.1 \( \text{set}(Ix \oplus (p_1, \ldots, p_k)) \cap F \neq \emptyset \)

PROVE: False

(8.1) Choose some \( j \in Ix \) such that \( p_j \in F \) and \( \text{eval}(\sigma'_{n}[\phi](bx_{j})) = \tau \)

PROOF: By (6.3) and assumption 4.1.

(8.2) Choose
\[
g_0 \in Q_0, \\
\sigma_0, \sigma_1 \in \hat{\Sigma}_T, \text{ and } \\
u \in \hat{E}^*
\]
such that
\[
[q_1, \sigma_1] \xrightarrow{\tau} [q_0, \sigma_0] \xrightarrow{\tau} [q_j, \sigma_1] \in EG(SM) \text{ and } \\
\text{comp}([SM], u, sp')
\]

PROOF: By assumption 1-3, (5)4, and Lemma 3.2.

(8.3) Choose \( \sigma_2 \in \hat{\Sigma}_T \) such that \( [q_j, \sigma_1] \xrightarrow{\tau} [p_j, \sigma_2] \)
(9.1) $\text{eval}(\sigma_1[\phi](e')) = e$

**Proof:** By (6.2).

(9.2) $\text{eval}(\sigma_1[\phi](bx_j)) = t$

**Proof:** By (8.2), (5.4), (6.1), (8.1) (since $q_j \xrightarrow{(e', bx_j, as_j)} p_j \in \mathcal{R}$), (8.2), assumption 2, and definition of side effect freedom (Def. 25).

(9.3) Q.E.D.

**Proof:** By (9.1), (9.2), and definition of the execution graph for state machines (Def. 11).

(8.4) $u \sim \langle e \rangle \in [SM]$

**Proof:** By (8.1) (since $p_j \in \mathcal{F}$) and (8.2) and (8.3) (since $[q_1, \sigma_1] \xrightarrow{(e)\sim \langle e \rangle} [p_j, \sigma_2]$), and definition of $\ldots$ (Def. 12).

(8.5) $u \sim \langle e \rangle \triangleleft s$

**Proof:** We know that $u \triangleleft sp'$ (by (8.2)). By definition of $\triangleleft$, this means that $u \sim \langle e \rangle \triangleleft sp' \sim \langle e \rangle$. Since $sp' \sim \langle e \rangle \subseteq s$ (by assumptions 2 and 3) we know that $u \sim \langle e \rangle \triangleleft s$ by definition of $\triangleleft$.

(8.6) Q.E.D.

**Proof:** (8.4) and (8.5) contradict assumption 1.1, therefore assumption 4.1 does not hold.

(7.2) Q.E.D.

**Proof:** By contradiction.

(6.6) $St(Q_n, e', Ix) \cap \mathcal{F} = \emptyset$ (where $St$ is the function defined in (Def. 24)).

(7.1) $Vi(Q_n, e', Ix) = \{q, (e', bx_1, as_1), p_1), \ldots, (q_n, (e', bx_n, as_n), p_n)\}$

**Proof:**

$$Vi(Q_n, e', Ix) = set(Ix \oplus ((q_1, (e', bx_1, as_1), p_1), \ldots, (q_n, (e', bx_n, as_n), p_n)))$$

By Def. 24

$$Vi(Q_n, e', Ix) = set(Ix \oplus list(Vi(Q_n, e')))$$

By Def. 24

$$Vi(Q_n, e', Ix) = set(Ix \oplus list(\{(q, (e'', bx, as), q') \in \mathcal{R} \mid q \in Q_n \land e = e''\}))$$

By Def. 24

(7.2) $St(Q_n, e', Ix) = set(Ix \oplus (p_1, \ldots, p_k))$

**Proof:**

$$St(Q_n, e', Ix) = \{q \in Q \mid \exists q' \xrightarrow{(e', bx', as')} q'' \in Vi(Q_n, e', Ix) : q = q''\}$$

By Def. 24

$$St(Q_n, e', Ix) = \{q \in Q \mid \exists q' \xrightarrow{(e', bx', as')} q'' \in set(Ix \oplus ((q_1, (e', bx_1, as_1), p_1), \ldots, (q_k, (e', bx_k, as_k), p_k))) : q = q''\}$$

By (7.1)

$$St(Q_n, e', Ix) = set(Ix \oplus (p_1, \ldots, p_k))$$

By set abstraction

(7.3) Q.E.D.

**Proof:** By (6.5) and (7.2).

(6.7) Q.E.D.
Proof: By (6) and definition of $ph2''$ (Def. 24), we have that
\[ Q_n \xrightarrow{(e', bx'', sa)} Q_n \cup St(Q_n, e', Ix) \in R' \] for some assignment sequence $sa$ ($sa$ is not important in the current argument). This means that $[Q_n, \sigma_n'] \xrightarrow{s_1} [Q_{n+1}, \sigma'_{n+1}] \in EG(SM')$ (where $Q_{n+1} = Q_n \cup St(Q_n, e', Ix)$ by (6) (since $\phi(e') = e$) and (6) (since eval($\sigma'_n[\phi]$) = $t$) and definition of the execution graph of state machines (Def. 11).

(5)6. Q.E.D.
Proof: By (5)1 - (5)5 and definition of $[]$ (Def. 12).

(4)3. Q.E.D.
Proof: By (4)1, (4)2, and induction over the length of $sp$.

(3)2. Q.E.D.
Proof: By $\forall$-rule.

(2)2. Q.E.D.
Proof: By (2)1.

(1)2. Assume: $1.1 s \in [SM']$
Proof: $(\forall t \in [SM]: \neg(t \lhd s))$

(2)1. Assume: $2.1 t \lhd s$ for some $t \in [SM]$
Proof: False

(3)1. Choose
\[
t' = (e_1, \ldots, e_n) \in \bar{E}^*,
q_0, q_1, \ldots, q_{n-1} \in Q \setminus F,
q_n \in F,
\sigma_1, \sigma_0, \sigma_1, \ldots, \sigma_n \in \bar{\Sigma},\text{ and}
\text{such that}
\]
\[
t' \lhd s,
\neg(\exists t'' \in [SM] : t'' \sqsubseteq t' \land t'' \lhd s), \text{and}
[q_1, \sigma_1] \xrightarrow{s_1} [q_0, \sigma_0] \xrightarrow{s_1} [q_1, \sigma_1] \xrightarrow{s_2} [q_2, \sigma_2] \ldots \xrightarrow{s_n} [q_n, \sigma_n] \in EG(SM)
\]
Proof: By assumption 1, assumption 2.1 and definition of $[]$ (Def. 12).

(3)2. Choose
\[
u \in \bar{E}^*
\text{such that}
\]
\[
(A) \ u \lhd (e_n) \sqsubseteq s \text{ and}
(B) \ \text{eval}([SM], (e_1, e_2, \ldots, e_{n-1}), u)
\]
(5)1. $pr([SM], (e_1, e_2, \ldots, e_{n-1})) \neq \emptyset$
Proof: By (3)1 and definition of $pr$ (Def. 25).

(5)2. Choose $u \in E^*$ such that $u \lhd (e_n) \sqsubseteq s$, $(e_1, e_2, \ldots, e_{n-1}) \lhd u$, and
\[
\neg(t' \lhd u).
\]
Proof: By assumption 2.1 and (3)1, we know that $(e_1, \ldots, e_n) \lhd s$. Choosing a prefix $u$ of $s$ such that (5)2 is satisfied, is therefore possible.

(5)3. Q.E.D.
Proof: By (5)1, (5)2, (3)1, and definition of $comp(\_ \_ \_)$ (Def. 25).

(3)3. Choose
\[
Q_1, Q_2, Q_3 \in Q', \text{ and}
\sigma'_1, \sigma'_2, \sigma'_3 \in \bar{\Sigma}
\text{such that}
\]
\[
[q_1, \sigma'] \xrightarrow{s_1} [Q_1, \sigma'_1] \xrightarrow{s_2} [Q_2, \sigma'_2] \xrightarrow{s_3} [Q_3, \sigma'_3] \in EG(SM')
\]
(4)1. $u \lhd (e_n) \in [SM']$
PROOF: By (3.2), we have that \( u \sim \langle e_n \rangle \sqsubseteq s \). Furthermore, by assumption 1.1, we know that \( s \in \llbracket SM' \rrbracket \). Since the semantic trace set of \( SM' \) is prefix-closed (because all states of \( SM' \) are final states), it is easy to see that \( u \sim \langle e_n \rangle \in \llbracket SM' \rrbracket \).

(4.2) Choose
\[
\begin{align*}
&u' \in (E \cup \{\epsilon\})^*, \\
&\sigma'_3 \in \Sigma, \text{ and } Q_3 \in Q'
\end{align*}
\]
such that
\[
[q'_1, \sigma_1] \xrightarrow{u'} [Q_3, \sigma'_3] \in EG(SM'), \text{ and }
\]
\[
u'_{E} = u \sim \langle e_n \rangle.
\]
PROOF: By (4.1) and definition of \( \llbracket \cdot \rrbracket \) (Def. 12).

(4.3) \( u' = \langle e \rangle \sim u \sim \langle e_n \rangle \)

PROOF: By assumption 1, \( SM \) is assumed to satisfy syntax constraints \( SM2 \) and \( SM3 \) that ensure that all transitions of \( SM \) except the outgoing transition for the initial state of \( SM \) are labeled by actions containing events. By definition of \( ph2'' \) (Def. 24) this means that the same constraints hold for the inverted state machine \( SM' \). Therefore we can assert that \( u' = \langle e \rangle \sim u \sim \langle e_n \rangle \) by (4.2).

(4.4) Q.E.D.

PROOF: By (4.2) and (4.3).

(3.4) \( q_{n-1} \in Q_2 \)

PROOF: By assumption 1-3, (3.1), (3.3), and Lemma 3.1.

(3.5) Choose
\[
e' \in \mathcal{E} \text{ and } \phi \in \hat{\Sigma}
\]
such that
\[
\text{Dom}(\phi) = pvar(e'),
\]
\[
\hat{\phi}(e') = e_n
\]
PROOF: By (3.1), and definition of the alphabet of state machines (Def. 10).

(3.6) Choose
\[
\text{bx} \in \text{BExp} \text{ and }
\]
\[
as \in (\text{Var} \times \text{Exp})^*
\]
such that
\[
q_{n-1} \xrightarrow{(e', \text{bx}, \text{as})} q_n \in \mathcal{R} \text{ and }
\]
\[
eval(\sigma_{n-1}[\phi](\text{bx})) = t
\]
PROOF: By (3.1), (3.5), and definition of the execution graph of state machines (Def. 11).

(3.7) \( \text{eval}(\sigma_2'[\phi](\text{bx})) = I \)

PROOF: By (3.3), (3.5), and definition of the execution graph of state machines (Def. 11).

(4.1) Choose \( Q_2 \) such that
\[
eval(\sigma_2'[\phi](\text{bx}_2')) = t
\]
PROOF: By (3.3), (3.5), and definition of the execution graph of state machines (Def. 11).

(4.2) \( \text{bx}_2' = conj(b, neg(disj(b_1, \ldots, bx, \ldots, b_k)) \text{ for some } b, b_1, \ldots, b_k \in \text{BExp} \)

(5.1) Choose \( Ix \in \mathbb{P}(\mathbb{N}) \) such that \( Q_3 = Q_2 \cup St(Q_2, e', Ix, V_2) \)

PROOF: By (4.1) and definition of \( ph2'' \) (Def. 24).

(5.2) Choose
\[
n_1', \ldots, n_m' \in Q
\]
\[
p_1', \ldots, p_m' \in Q
\]
\[
\text{bx}_1, \ldots, \text{bx}_m \in \text{BExp}
\]
\(a_{s_1}, \ldots, a_{s_m} \in (\text{Var} \times \text{Exp})^*\)

such that
\[Vi(Q_2, e', Ix) = \text{set}(Ix \oplus (q'_1, (e', bx_1, a_{s_1}), p'_1), \ldots, (q'_m, (e', bx_m, a_{s_m}), p'_m))\]

where \(Vi\) is the function defined in Def. 24.

**Proof:** By (4)1 and definition of \(Vi\) (Def. 24).

(5.3) Choose \(j \in \mathbb{N}\) such that
\[(q'_j, (e', bx_j, a_{s_j}), p_j) = (q_{n-1}, (e', bx, as), q_n)\]

**Proof:** By (5)2, (3)4 (since \(q_{n-1} \in Q_2\)), and (3)6 (since \(q_{n-1} \langle e', bx, as\rangle q_n \in R\)).

(5.4) \(j \notin Ix\)

(6.1) Assume: 3.1 \(j \in Ix\)

**Prove:** False

(7.1) \(St(Q_2, e', Ix) \cap F \neq \emptyset\) where \(St\) is the function defined in Def. 24.

**Proof:** By (5)2, (5)3, assumption 3.1, and definition of \(St\) (Def. 24), we have that \(p_j \in St(Q_2, e', Ix)\). By (5)3 we know that \(p_j = q_n\). Since \(q_n \in F\) (by (3)1), this implies that \(p_j \in F\) which again implies that (7)1 holds.

(7.2) Q.E.D.

**Proof:** (7)1 contradicts (4)1 by definition of \(ph2''\) (Def. 24).

(6.2) Q.E.D.

By (6)1, \(j \notin Ix\) leads to a contradiction, therefore \(j \notin Ix\).

(5.5) \(bx_j \in \text{set}(\mathbb{N} \setminus Ix \oplus (bx_1, \ldots, bx_m))\)

**Proof:** By (5)2, (5)4 and definition of \(\oplus\) and \(\text{set}\) (see App. D.1.2).

(5.6) \(bx'_j = \text{conj}(Ix \oplus (bx_1, \ldots, bx_m), \text{neg}(\text{disj}(\mathbb{N} \setminus Ix \oplus (bx_1, \ldots, bx_m))))\)

**Proof:** By (4)1, (5)2 and definition of \(ph2''\) (Def. 24).

(5.7) Q.E.D.

**Proof:** By (5)5 and (5)6.

(4.3) Q.E.D.

**Proof:** By (4)1, (4)2, and definition of \(\text{conj}, \text{disj},\) and \(\text{neg}\).

(3.8) \(\text{eval}(\sigma_{n-1}[\phi](bx)) = \text{eval}(\sigma'_{n-1}[\phi](bx))\)

(4.1) \(\sigma_{n-1} \cap (\text{var}(bx) \times \text{Exp}) = \sigma'_{n-1} \cap (\text{var}(bx) \times \text{Exp})\)

(5.1) \([q_1, \sigma_1] \xrightarrow{\text{EG}} [q_0, \sigma_0] \xrightarrow{\sigma_0} [q_1, \sigma_1] \xrightarrow{\sigma_2} [q_2, \sigma_2] \cdots \xrightarrow{\sigma_{n-1}} [q_{n-1}, \sigma_{n-1}] \in \text{EG}(SM)\)

**Proof:** By (3)1.

(5.2) \([q'_1, \sigma'_1] \xrightarrow{\text{EG}} [Q_1, \sigma'_1] \xrightarrow{\sigma'_0} [Q_2, \sigma'_2] \in \text{EG}(SM')\)

**Proof:** By (3)3.

(5.3) \(q_{n-1} \langle e', bx, as\rangle q_n \in R\)

**Proof:** By (3)6.

(5.4) \(\text{comp}([SM], (e_1, e_2, \ldots, e_{n-1}), u)\)

**Proof:** By (3)2.

(5.5) Q.E.D.

**Proof:** By (5)1 - (5)4, assumption 2, and definition of side-effect freedom (Def. 25).

(4.2) \(\sigma_{n-1}[\phi] \cap (\text{var}(bx) \times \text{Exp}) = \sigma'_{n-1}[\phi] \cap (\text{var}(bx) \times \text{Exp})\)

**Proof:** By (4)1 and definition of the data state overriding operator (4)3. Q.E.D.
Lemma 3.1 The proof is based on induction. We make use of the predicate \( \text{Ind} \in \text{SM} \times \text{SM} \times \mathbb{N} \to \mathbb{B} \) which highlights the induction:

\[
\text{Ind}((\mathcal{E}, Q, R, q_1, \mathcal{F}), (\mathcal{E}', Q', R', q'_1, \mathcal{F}'), m) \equiv \\
\forall n \in \mathbb{N} : \\
\forall q_1 \in Q \setminus \mathcal{F} : \cdots : \forall q_n \in Q \setminus \mathcal{F} : \\
\forall \sigma_1 \in \hat{T}_R : \cdots : \forall \sigma_n \in \hat{T}_R : \\
\forall Q_1, Q_n \in Q' : \\
\forall \sigma'_1 \in \hat{T}_R : \forall \sigma'_n \in \hat{T}_R : \\
\forall e_1 \in E : \cdots : \forall e_n \in E : \forall u \in E^* \\
\wedge n \leq m \\
\wedge \text{comp}(\text{SM}, (e_1, \ldots, e_{n-1}), u) \\
\wedge [q_1, \sigma_1] \xrightarrow{e_1} [q_1, \sigma_1] \xrightarrow{e_2} [q_2, \sigma_2] \cdots \\
\xrightarrow{e_n} [q_n, \sigma_n] \in \text{EG}((\mathcal{E}, Q, R, q_1, \mathcal{F})) \\
\wedge [q'_1, \sigma'_1] \xrightarrow{u} [Q_1, \sigma'_1] \xrightarrow{u} \\
[Q_n, \sigma'_n] \in \text{EG}((\mathcal{E}', Q', R', q'_1, \mathcal{F}')) \\
\implies q_n \in Q_n
\]

Assume: 1. \( SM = (\mathcal{E}, Q, R, q_1, \mathcal{F}) \) for \( SM \in \text{SM} \) satisfying SM1 - SM4 and \( \langle \rangle \notin \llbracket SM \rrbracket \)
2. \( ph2'' \) is side effect free for \( SM \)
3. \( SM' = ph2''(SM) = (\mathcal{E}', Q', R', q'_1, \mathcal{F}') \)
4. \( n \in \mathbb{N} \)
5. \( q_1, q_n, \sigma_1, \ldots, \sigma_n \in \hat{T}_R \)
6. \( Q_1, Q_n \in Q' \)
7. \( \sigma'_1, \sigma'_n \in \hat{T}_R \)
8. \( e_1, \ldots, e_n \in E \)
10. \( \text{comp}(\llbracket SM \rrbracket, (e_1, \ldots, e_n), u) \) for \( u \in E^* \)
11. \( [q_1, \sigma_1] \xrightarrow{e_1} [q_1, \sigma_1] \xrightarrow{e_2} [q_2, \sigma_2] \cdots \xrightarrow{e_n} [q_n, \sigma_n] \in \text{EG}(SM) \)
12. \( [q'_1, \sigma_1] \xrightarrow{u} [Q_1, \sigma'_1] \xrightarrow{u} \\
[Q_n, \sigma'_n] \in \text{EG}(SM') \)

Prove: \( q_n \in Q_n \)

(1). Case: 1. \( n = 0 \)

Proof: Case assumption 1.1 leads to a contradiction. To see this, note that by case assumption 1.1, assumption 11, and definition of \( \llbracket \cdot \rrbracket \) (Def. 12) have \( \langle \rangle \notin \llbracket SM \rrbracket \) which contradicts assumption 1.

(1). Case: 1. \( \text{Ind}(SM, SM', n - 1) \)

(2). Case: 1. \( \text{Ind}(SM, SM', n - 1) \)

(4). Case: 1. \( \text{Ind}(SM, SM', n - 1) \)

Proof: By assumption 10 and definition of \( pr \) (Def. 25).
(4)2. Choose \( u' \in E^* \) such that \( u' \prec (e_n) \cup u \) and \( \langle e_1, \ldots, e_{n-1} \rangle \prec u' \) and \( \neg \phi(e_n) \).

Proof: By assumption 10 and definition of \( \text{comp}_n(\omega, \omega) \) (Def. 25).

(4)3. \( \neg \exists s' \in \text{pr}(\langle SM \rangle, \langle e_1, \ldots, e_{n-1} \rangle) : \langle e_1, \ldots, e_{n-1} \rangle \cup s' \prec u' \).

Proof: By (4)2.

(4)4. Q.E.D.

Proof: By (4)1 - (4)3 and definition of \( \text{comp}_n(\omega, \omega) \) (Def. 25).

(2)2. Choose \( Q_{n-2}, Q_{n-1} \in Q' \) and \( \sigma'_{n-2}, \sigma'_{n-1} \in \Sigma_T \) such that \( [q_i', \sigma_i] \xrightarrow{\langle e_n \rangle} [Q_{n-2}, \sigma'_{n-2}] \xrightarrow{\langle e_n \rangle} [Q_{n-1}, \sigma'_{n-1}] \in EG(SM') \).

Proof: By assumption 12, (2)1 and definition of the execution graph for state machines (Def. 11).

(2)3. \( q_{n-1} \in Q_{n-2} \).

Proof: By (2)1, (2)2, assumptions 4-12, and induction hypothesis 1.1.

(2)4. Choose

\( (e'_n, bx, as) \in \text{Act} \) and

\( \phi \in \Sigma \)

such that

\( q_{n-1} \xrightarrow{(e'_n, bx, as)} q_n \in R, \)

\( \text{Dom}(\phi) = \text{var}(e'_n), \)

\( \phi(e'_n) = e_n, \) and

\( \text{eval}(\sigma_{n-1}[\phi](bx)) = t \)

Proof: By assumption 11 and definition of the execution graph for state machines (Def. 11).

(2)5. Choose

\( (e'_n, bx', as') \in \text{Act} \) and

such that

\( Q_{n-2} \xrightarrow{(e'_n, bx', as')} Q_{n-1} \in R', \)

\( \text{eval}(\sigma'_{n-1}[\phi](bx')) = t \)

Proof: By (2)4 (since \( \text{Dom}(\phi) = \text{var}(e'_n) \) and \( \phi(e'_n) = e_n \)) assumption 12, (2)2, and definition of the execution graph for state machines (Def. 11).

(2)6. \( q_n \in Q_{n-1} \).

(3)1. Choose \( Ix \in P(N) \) such that \( Q_{n-1} = Q_{n-2} \cup St(Q_{n-2}, e'_n, Ix) \) where

\( St \) is the function defined in Def. 24.

Proof: By (2)5 and definition of \( \phi h2'' \) (Def. 24).

(3)2. Choose

\( q'_1, \ldots, q'_m \in Q_{n-2} \)

\( p'_1, \ldots, p'_m \in Q \)

\( bx_1, \ldots, bx_m \in \text{BExp} \)

\( as_1, \ldots, as_m \in (\text{Var} \times \text{Exp})^* \)

such that

\( Vi(Q_{n-2}, e'_n, Ix) = \)

\( \text{set}(Ix \oplus ((q'_1, (e'_n, bx_1, as_1), p'_1), \ldots, (q'_m, (e'_n, bx_m, as_m), p'_m))) \)

where \( Vi \) is the function defined in Def. 24.

Proof: By definition of \( Vi \) (Def. 24).

(3)3. Choose \( j \in \mathbb{N} \) such that

\( (q'_j, bx_j, as_j) = (q_{n-1}, (e'_n, bx, as), q_n) \)

Proof: By (2)3, (2)4, (3)2 and definition of \( Vi \) (Def. 24).

(3)4. \( j \in Ix \).
Lemma 3.2

The proof is based on induction. We make use of the predicate \( \text{Ind} \in \text{SM} \times \text{SM} \times \text{E}^* \rightarrow \mathbb{B} \) which highlights the induction. Let \( \text{SM} = \ldots \)
\( (E, Q, R, q_1, F) \), and \( SM' \equiv (E', Q', R', q'_1, F') \), then \( Ind \) is defined by

\[
Ind(SM, SM', s) \equiv \\
\forall s' \in E^* : \\
\forall Q_n \in Q' : \forall q_n \in Q \cap Q_n : \\
\forall \sigma_I \in \widehat{\Sigma}_T : \forall \sigma'_n \in \widehat{\Sigma}_T : \\
\wedge s' \subseteq s \\
\wedge [q_I, \sigma_I] \xrightarrow{(c,s)} [Q_n, \sigma'_n] \in EG(SM') \\
\implies \exists u \in E^* : \\
\exists \sigma_n \in \widehat{\Sigma}_T : \\
\wedge [q_I, \sigma_I] \xrightarrow{(c,s)} [Q_n, \sigma_n] \in EG(SM) \\
\wedge \text{comp}(\llbracket SM \rrbracket, u, s')
\]

Assume: 1. \( SM = (E, Q, R, q_1, F) \) for \( SM \in SM \) satisfying \( SM1 - SM4 \) and \\
(\( \emptyset \not\in \llbracket SM \rrbracket \)) 2. \( ph2'' \) is side effect free for \( SM \) 3. \( SM' = ph2''(SM) = (E', Q', R', q_1, F') \) 4. \( s \in E^* \) 5. \( Q_n \in Q' \) 6. \( \sigma_I, \sigma'_n \in \widehat{\Sigma}_T \) 7. \( q_n \in Q_n \cap Q \) 8. \( [q'_I, \sigma_I] \xrightarrow{(c,s)} [Q_n, \sigma'_n] \in EG(SM') \)

Prove: \( \exists u \in E^* : \\
\exists \sigma_n \in \widehat{\Sigma}_T : \\
\wedge [q_I, \sigma_I] \xrightarrow{(c,u)} [Q_n, \sigma_n] \in EG(SM) \\
\wedge \text{comp}(\llbracket SM \rrbracket, u, s) \)

(1.1) Case: 1.1. \( s = \emptyset \)

(2.1) Choose \( q_I \xrightarrow{(c,s,a)} q' \in R \)

Proof: By assumption 1 since \( SM \) satisfies syntax constraint \( SM2 \).

(2.2) \( q'_I \xrightarrow{(c,s,a)} \{ q' \} \in R' \)

Proof: By (2.1), assumption 3, and definition of \( ph2'' \) (Def. 24).

(2.3) \( [q'_I, \sigma_I] \xrightarrow{(c,s)} [Q_n, \sigma'_n] \in EG(SM') \)

Proof: By assumption 8, case assumption 1.1, and assumptions 1 (since \( SM \) is assumed to satisfy syntax constraint \( SM3 \)) and 3 and definition of \( ph2'' \) (Def. 24).

(2.4) \( q_n = q' \)

Proof: By assumption 7, (2.2) and (2.3) and definition of \( ph2'' \) (Def. 24) (since \( Q_n = \{ q' \} \)).

(2.5) Choose \( \sigma_n \in \widehat{\Sigma}_T \) such that \( [q_I, \sigma'_I] \xrightarrow{(c,s)} [q_n, \sigma_n] \in EG(SM) \)

Proof: By (2.1), (2.4) and definition of the execution graph for state machines (Def. 11).

(2.6) Q.E.D.

Proof: By (2.5).

(1.2) Case: 1.1. \( s = s' \sim (c) \)

1.2 \( Ind(SM, SM', s') \)

(2.1) Choose \( Q_{n-1} \in Q' \) and \( \sigma'_{n-1} \in \widehat{\Sigma}_T \) such that \( [q'_I, \sigma'_I] \xrightarrow{(c,s')} [Q_{n-1}, \sigma'_{n-1}] \xrightarrow{e} [Q_n, \sigma'_n] \in EG(SM') \)
PROOF: By assumption 8, assumption 1.1, and definition of the execution graph for state machines (Def. 11).

(2)2. Choose 
\[(e', bx', as') \in \text{Act} \text{ and} \]
\[\phi \in \Sigma \]
such that
\[Q_{n-1}^{(e',bx',as')} Q_n \in R'\]
\[\text{Dom}(\phi) = \text{var}(e'), \]
\[\phi(e') = e, \text{ and} \]
\[\text{eval}(\sigma'_{n-1}[\phi](bx')) = t\]

PROOF: By (2)1 and definition of the execution graph for state machines (Def. 11).

(2)3. CASE: 2.1 \(\exists x \in \mathbb{N} : q_n \in St(Q_{n-1}, e', Ix)\)

(3)1. Choose \(Ix \in \mathbb{N}\) and \(q_{n-1}^{(e',bx,sa)} q_n \in R\) such that \(q_{n-1}^{(e',bx,sa)} q_n \in Vi(Q_{n-1}, e', Ix)\)

PROOF: By case assumption 2.1 and definition of Vi (Def. 24).

(3)2. Choose 
\[\sigma_{n-1} \in \hat{\Sigma}_T \text{ and,}\]
\[u' \in E' \]
such that
\[(A) [q_I, \sigma_I] \xrightarrow{(e)-u'} [q_{n-1}, \sigma_{n-1}] \in EG(SM),\]
\[(B) \text{comp}([\text{SM}], u', s')\]

PROOF: By assumptions 1.1, 1.2, and (3)1.

(3)3. Choose \(\sigma_n \in \hat{\Sigma}_T\) such that
\[\sigma_{n-1} = [q_n, \sigma_n] \in EG(SM)\]

PROOF: By (2)1, (2)2, (3)2, assumption 2, and assumption 3.

(3)4. \(\text{comp}([\text{SM}], u' \sim (e), s' \sim (e))\)

(4)1. \(\text{pr}([\text{SM}], u' \sim (e)) \neq \emptyset\)

PROOF: By (3)2 and definition of \(\text{pr}\) (Def. 25).

(4)2. \(u' \sim (e) \prec s' \sim (e)\)

PROOF: By (3)2 and definition of \(\text{comp}(-, -)\) (Def. 25).

(4)3. \(\neg \exists s'' \in \text{pr}([\text{SM}], u' \sim (e)) : u' \sim (e) \sqcup s'' \prec s' \sim (e)\)

PROOF: If (4)3 does not holds, then we can choose \(e' \in \mathcal{E}\) such that
\[u' \sim (e, e') \prec s' \sim (e)\]
but by definition of \(\prec\), this is impossible.

(4)4. Q.E.D.

PROOF: By (4)1 - (4)3 and definition of \(\text{comp}(-, -)\) (Def. 25).

(3)5. Q.E.D.

PROOF: By (3)2 - (3)4.

(2)4. CASE: 2.1 \(\neg(\exists x \in \mathbb{N} : q_n \in St(Q_{n-1}, e', Ix))\)

(3)1. Choose 
\[q_{n-1} \in Q_{n-1} \cap Q,\]
\[\sigma_{n-1} \in \hat{\Sigma}_T \text{ and,}\]
\[u' \in E' \]
such that
\[(A) [q_I, \sigma_I] \xrightarrow{(e)-u'} [q_{n-1}, \sigma_{n-1}] \in EG(SM),\]
\[(B) \text{comp}([\text{SM}], u', s'),\]
\[(C) q_{n-1} = q_n\]
Proof: (A) and (B) hold by case assumption 1.2. (C) holds by case assumption 7, assumption 2.1, and definition of ph2" (Def. 24) since we have Qn = Qn-1.

(3.2) \([n_{n-1}, \sigma_{n-1}] \Rightarrow [n_n, \sigma_{n-1}] \in EG(SM)\)
Proof: By (3.1) and definition of the execution graph for state machines (Def. 11).

(3.3) \(\text{comp}(\llbracket SM \rrbracket, u', s' \sim (e))\)

(4.1) \(pr(\llbracket SM \rrbracket, u') \neq \emptyset\)
Proof: By (3.1) and definition of pr (Def. 25).

(4.2) \(u' \prec s' \sim (e)\)
Proof: By (3.1) and definition of \(\text{comp}(\cdot, \cdot)\) (Def. 25).

(4.3) \(\neg(\exists s'' \in pr(\llbracket SM \rrbracket, u') : u' \sim (e) \sqsubseteq s'' \land s'' \prec u' \sim (e))\)
Proof: If (4.3) holds, then there must be trace \(t \in \llbracket SM \rrbracket\) such that \(u' \sim (e) \sqsubseteq t\). However, this contradicts case assumption 2.1.

(4.4) Q.E.D.
Proof: By (4.1) - (4.3) and definition of \(\text{comp}(\cdot, \cdot)\) (Def. 25).

(4.4) Q.E.D.
Proof: By (3.1) - (3.3).

(2.5) Q.E.D.
Proof: By (2.1) - (2.4).

(1.3) Q.E.D.
Proof: By (1.1) and (1.2) and induction.

Corollary 1 Let SM be a well formed state machine such that \(\langle \rangle \notin \llbracket SM \rrbracket\) and ph2 be side effect free for SM, then ph2(SM) is an inversion of SM, i.e.,

\(\text{inv}(SM, ph2(SM))\)

Proof of Corollary 1

Assume: 1. \(SM = (E, Q, R, q_I, F)\) for \(SM \in SM\) satisfying SM1 - SM4 and \(\langle \rangle \notin \llbracket SM \rrbracket\)
2. ph2 is side effect free for SM
3. \(SM' = ph2(SM) = (E', Q', R', q_I, F')\)
4. \(s \in \{e \in \bar{E} \mid \exists e' \in E: e = e'\}^*\)

Proof: \((\forall t \in \llbracket SM \rrbracket: \neg(t \sim s)) \iff s \in \llbracket SM' \rrbracket\)

(1.1) Assume: 1.1. \((\forall t \in \llbracket SM \rrbracket: \neg(t \sim s))\)
Proof: \(s \in \llbracket SM' \rrbracket\)

(2.1) \(\forall sp \in \bar{E}^*: sp \sqsubseteq s \implies sp \in \llbracket SM' \rrbracket\)

(3.1) Assume: 2.1 \(sp \sqsubseteq s\) for some \(sp \in \bar{E}^*\)
Proof: \(sp \in \llbracket SM' \rrbracket\)

(4.1) Case: 3.1. \(sp = \langle \rangle\)
Proof: \(q_I' \in F'\) by definition of ph2 (Def. 26) and assumption 3. This means that \(\langle \rangle \in \llbracket SM' \rrbracket\) by definition of \([\cdot]\) (Def. 12).

(4.2) Case: 3.1. \(sp = sp' \sim (e)\) for some \(sp' \in \llbracket SM' \rrbracket\) (the induction hypothesis) and \(e \in \bar{E}\)

(5.1) Choose \(e_1, e_2, \ldots, e_n \in \bar{E}\) such that \(sp' = (e_1, e_2, \ldots, e_n)\)
Proof: By assumption 2.1 and assumption 3.1.
(5.2) Choose $t \in (E \cup \{\epsilon\})^*$, $(Q_n, V_n) \in \mathcal{Q}'$ and $\sigma_1, \sigma'_n \in \tilde{\Sigma}_T$ such that
\[ [q'_1, \sigma_1] \xrightarrow{t} [(Q_n, V_n), \sigma'_n] \in EG(SM') \text{ and } t|_E = sp' \]

**PROOF:** By case assumption 3.1 (since $sp' \in [SM']$) and definition of $[\cdot]$ (Def. 12).

(5.3) $t = (\epsilon) \sim sp'$

**PROOF:** By assumption 1, $SM$ is assumed to satisfy syntax constraints $SM_2$ and $SM_3$ that ensure that all transitions of $SM$ except the outgoing transition for the initial state of $SM$ are labeled by actions containing events. By definition of $ph_2$ (Def. 26) this means that the same constraints hold for the inverted state machine $SM'$. Therefore we can assert that $t = (\epsilon) \sim sp'$.

(5.4) Choose
\[ (Q_0, V_0), (Q_1, V_1), (Q_2, V_2), \ldots, (Q_{n-1}, V_{n-1}) \in \mathcal{Q}' \text{ and } \sigma'_0, \ldots, \sigma'_{n-1} \in \tilde{\Sigma}_T \]
such that
\[ [q'_1, \sigma_1] \xrightarrow{t} [(Q_0, V_0), \sigma'_0] \xrightarrow{t} [(Q_1, V_1), \sigma'_1] \xrightarrow{t} [(Q_2, V_2), \sigma'_2] \cdots \xrightarrow{t} [(Q_n, V_n), \sigma'_n] \in EG(SM') \]

**PROOF:** By (5.1), (5.2), and (5.3).

(5.5) Choose $(Q_{n+1}, V_{n+1}) \in \mathcal{F}'$ and $\sigma'_{n+1} \in \tilde{\Sigma}_T$ such that
\[ [(Q_n, V_n), \sigma'_n] \xrightarrow{t} [(Q_{n+1}, V_{n+1}), \sigma'_{n+1}] \in EG(SM') \]

(6.1) Choose
\[ q_1, \ldots, q_k \in Q_n, \]
\[ p_1, \ldots, p_k \in Q, \]
\[ e' \in E, \]
\[ bx_1, \ldots, bx_k \in B\text{Exp}, \text{ and} \]
\[ as_1, \ldots, as_k \in (\text{Var} \times \text{Exp})^* \]
such that
\[ ((q_1, (e', bx_1, as_1), p_1), \ldots, (q_k, (e', bx_k, as_k), p_k)) = \text{list}(((q, (e'', bx, as), q') \in R | q \in Q_n \land e = e'' \setminus V_n) \]

**PROOF:** By assumption 1, definition of the alphabet of state machines (Def. 10), and definition of the list function (see App. D.1.2).

(6.2) Choose $\phi \in \tilde{\Sigma}$ such that $\text{Dom}(\phi) = \text{pear}(e')$ and $\phi(e') = e$

**PROOF:** By (6.1) and definition of the alphabet of state machines (Def. 10).

(6.3) Choose $Ix \subseteq \{1, \ldots, n\}$ such that $i \in Ix$ iff $\text{eval}(\sigma'_n[\phi](bx_i)) = t$

**PROOF:** Trivial.

**LET:** $bx = bx_1, bx_2, \ldots, bx_k$

**LET:** $bx'' = \text{conj}((\text{conj}(Ix \oplus bx), \text{neg}(\text{disj}((1, \ldots, n) \setminus Ix \oplus bx))))$

(6.4) $\text{eval}(\sigma'_n[\phi](bx'')) = t$

**PROOF:** By (6.3), definition of $bx''$, and definition of $\text{conj}$, $\text{disj}$, and $\text{neg}$ (Sect. D.1.2).

(6.5) $\text{set}(Ix \oplus (p_1, \ldots, p_k)) \cap \mathcal{F} = \emptyset$

(7.1) **ASSUME:** $4.1$ $\text{set}(Ix \oplus (p_1, \ldots, p_k)) \cap \mathcal{F} \neq \emptyset$

**PROVE:** False

(8.1) Choose some $j \in Ix$ such that $p_j \in \mathcal{F}$ and $\text{eval}(\sigma'_n[\phi](bx_j)) = t$

**PROOF:** By (6.3) and assumption 4.1.

(8.2) Choose
\( q_0 \in Q_0 \),
\( \sigma_0, \sigma_1 \in \Sigma_T \), and
\( u \in E^* \)
such that
\[ [q_I, \sigma_1] \xrightarrow{v} [q_0, \sigma_0] \xrightarrow{u} [q_j, \sigma_1] \in E(SM) \) and
\( \text{comp}(\text{SM}, u, sp') \)

**Proof:** By assumption 1-3, (5)4, and Corollary 3.2.

(8.3) Choose \( \sigma_2 \in \Sigma_T \) such that \( [q_j, \sigma_1] \xrightarrow{v} [p_j, \sigma_2] \)

(9.1) \( \text{eval}(\sigma_1[\phi](e')) = e \)

**Proof:** By (6)2.

(9.2) \( \text{eval}(\sigma_1[\phi](bx_j)) = t \)

**Proof:** By (8)2, (5)4, (6)1, (8)1 (since \( q_j \xrightarrow{\epsilon, bx_j, as_j} p_j \in R \)), (8)2, assumption 2, and definition of side effect freedom (Def. 25).

(9.3) Q.E.D.

**Proof:** By (9)1, (9)2, and definition of the execution graph for state machines (Def. 11).

(8.4) \( u \prec \langle e \rangle \in \text{SM} \)

**Proof:** By (8)1 (since \( p_j \in F \)) and (8)2 and (8)3 (since
\( [q_I, \sigma_1] \xrightarrow{\langle e \rangle, \sigma} [p_j, \sigma_2] \)), and definition of \( \text{SM} \) (Def. 12).

(8.5) \( u \prec \langle e \rangle \prec s \)

**Proof:** We know that \( u \prec sp' \) (by (8)2). By definition of \( \prec \), this means that \( u \prec \langle e \rangle \prec sp' \prec \langle e \rangle \). Since \( sp' \prec \langle e \rangle \subseteq s \) (by assumptions 2.1 and 3.1) we know that \( u \prec \langle e \rangle \prec s \) by definition of \( \prec \).

(8.6) Q.E.D.

**Proof:** (8)4 and (8)5 contradict assumption 1.1, therefore assumption 4.1 does not hold.

(7.2) Q.E.D.

**Proof:** By contradiction.

(6.6) \( St(Q_n, e', Ix, V_n) \cap F = \emptyset \) (where \( St \) is the function defined in (Def. 26)).

(7.1) \( Vi(Q_n, e', Ix, V_n) = \set{Ix \oplus ((q_1, (e', bx_1, as_1), p_1), \ldots, (q_n, (e', bx_n, as_n), p_n))} \)

**Proof:**

\[
Vi(Q_n, e', Ix, V_n) = \set{Ix \oplus \text{list}(Vi(Q_n, e', V_n))} \quad \text{By Def. 26}
\]
\[
= \set{Ix \oplus \text{list}(\set{(q, (e'' , bx, as), q') \in R \mid q \in Q_n \wedge e = e''}) \setminus V_n) \)] \quad \text{By Def. 26}
\]
\[
= \set{Ix \oplus \set{(q_1, (e', bx_1, as_1), p_1), \ldots, (q_k, (e', bx_k, as_k), p_k))} \quad \text{By (6)1}
\]

(7.2) \( St(Q_n, e', Ix, V_n) = \set{Ix \oplus (p_1, \ldots, p_k)} \)
Proof:

\[ \begin{align*}
S(t(Q_n, e', Ix, V_n)) &= \{ q \in Q \mid \exists q' (e', bx, as) \xrightarrow{t'} q'' \}
\leq V \langle Q_n, e', Ix, V_n \rangle : q = q'' \} \quad \text{By Def. 26} \\
&= \{ q \in Q \mid \exists q' (e', bx, as) \xrightarrow{t'} q'' \}
\subseteq set(Ix \oplus ((q_1, (e', bx_1, as_1), p_1), \ldots, (q_k, (e', bx_k, as_k), p_k)) : q = q'' \} \quad \text{By } (7)1 \\
&= set(Ix \oplus (p_1, \ldots, p_k)) 
\end{align*} \]

By set abstraction

(7)3. Q.E.D.

Proof: By (6)5 and (7)2.

(6)7. Q.E.D.

Proof: By (6)6, definition of ph2 (Def. 26), (6)2, and definition of the execution graph of state machines (Def. 11).

(5)6. Q.E.D.

Proof: By (5)1 - (5)5 and definition of \( \llbracket \) (Def. 12).

(4)3. Q.E.D.

Proof: By (4)1, (4)2, and induction over the length of sp.

(3)2. Q.E.D.

Proof: By \( \forall \)-rule.

(2)2. Q.E.D.

Proof: By (2)1.

(1)2. Assume: 1.1 \( s \in \llbracket SM' \rrbracket \)

Proof: \( \forall t \in \llbracket SM \rrbracket : \neg(t \circ s) \)

(2)1. Assume: 2.1 \( t \circ s \) for some \( t \in \llbracket SM \rrbracket \)

Proof: False

(3)1. Choose

\( t' = (e_1, \ldots, e_n) \in E^* \),

\( q_0, q_1, \ldots, q_{n-1} \in Q \setminus \mathcal{F} \),

\( q_n \in \mathcal{F} \),

\( \sigma_1, \sigma_0, \sigma_1, \ldots, \sigma_n \in \widehat{\mathcal{S}} \),

such that

\( t' \circ s \),

\( \neg(\exists I' \in \llbracket SM \rrbracket : I'' \sqsubseteq I'' \wedge t' \circ s) \), and

\( [q_1, \sigma_1] \xrightarrow{e_1} [q_0, \sigma_0] \xrightarrow{e_2} [q_1, \sigma_1] \xrightarrow{e_3} [q_2, \sigma_2] \ldots \xrightarrow{e_n} [q_n, \sigma_n] \in EG(SM) \)

Proof: By assumption 1, assumption 2.1 and definition of \( \llbracket \) (Def.12).

(3)2. Choose

\( u \in E^* \)

such that

(A) \( u \sqsubseteq (e_n) \sqsubseteq s \) and

(B) \( \text{comp}(\llbracket SM \rrbracket, (e_1, e_2, \ldots, e_{n-1}), u) \)

(5)1. \( p_r(\llbracket SM \rrbracket, (e_1, e_2, \ldots, e_{n-1})) \neq \emptyset \)

Proof: By (3)1 and definition of \( p_r \) (Def. 25).

(5)2. Choose \( u \in E^* \) such that \( u \sqsubseteq (e_n) \sqsubseteq s \), \( (e_1, e_2, \ldots, e_{n-1}) \circ u \), and

\( \neg(t' \circ u) \).

Proof: By assumption 2.1 and (3)1, we know that \( (e_1, \ldots, e_n) \circ u \).

Choosing a prefix \( u \) of such that (5)2 is satisfied, is therefore possible.
(5)3. Q.E.D.
Proof: By (5)1, (5)2, (3)1, and definition of \textit{comp}(\_\_\_\_) (Def. 25).

(3)3. Choose
\[(Q_1, V_1), (Q_2, V_2), (Q_3, V_3) \in \mathcal{Q}', \text{ and}
\sigma'_1, \sigma'_2, \sigma'_3 \overset{\Sigma}{\triangleright}
\]
such that
\[\varepsilon \overset{\sigma'_1}{\triangleright} [(Q_1, V_1), \sigma'_1] \overset{\sigma'_2}{\triangleright} [(Q_2, V_2), \sigma'_2] \overset{\epsilon n}{\triangleright} [(Q_3, V_3), \sigma'_3] \in \mathcal{EG}(SM')
\]

(4)1. \(u \sim \langle e_n \rangle \in [\mathcal{SM}']\)
Proof: By (3)2, we have that \(u \sim \langle e_n \rangle \subseteq s\). Furthermore, by assumption 1.1, we know that \(s \in [\mathcal{SM}']\). Since the semantic trace set of \(SM'\) is prefix-closed (because all states of \(SM'\) are final states), it is easy to see that \(u \sim \langle e_n \rangle \in [\mathcal{SM}']\).

(4)2. Choose \(u' \in (\mathcal{E} \cup \{\epsilon\})^*, \sigma'_3 \in \Sigma, \text{ and } (Q_3, V_3) \in \mathcal{Q}'\)
such that
\[\epsilon \overset{\sigma'_3}{\triangleright} [(Q_3, V_3), \sigma'_3] \in \mathcal{EG}(SM'), \text{ and}
\]
\[\varepsilon \mid \mathcal{E} = u \sim \langle e_n \rangle
\]
Proof: By (4)1 and definition of \([\_\_\_]\) (Def. 12).

(4)3. \(u' = \langle \epsilon \rangle \sim u \sim \langle e_n \rangle\)
Proof: By assumption 1, \(SM\) is assumed to satisfy syntax constraints \(SM2\) and \(SM3\) that ensure that all transitions of \(SM\) except the outgoing transition for the initial state of \(SM\) are labeled by actions containing events. By definition of \(ph2\) (Def. 26) this means that the same constraints hold for the inverted state machine \(SM'\). Therefore we can assert that \(u' = \langle \epsilon \rangle \sim u \sim \langle e_n \rangle\) by (4)2.

(4)4. Q.E.D.
Proof: By (4)2 and (4)3.

(3)4. \(q_{n-1} \in Q_2\)
Proof: By assumption 1-3, \(3)1, \(3)3, \text{ and Corollary 3.1.}

(3)5. Choose \(e' \in \mathcal{E} \text{ and } \phi \in \Sigma\)
such that
\[\text{Dom}(\phi) = \text{pvar}(e'),
\]
\[\phi(e') = e_n
\]
Proof: By (3)1, and definition of the alphabet of state machines (Def. 10).

(3)6. Choose \(bx \in \mathbf{BExp} \text{ and}
\)
\(\mathit{as} \in (\mathbf{Var} \times \mathbf{Exp})^*\)
such that
\[q_{n-1} \overset{(e', bx, as)}{\longrightarrow} q_n \in \mathcal{R} \text{ and}
\]
\[\text{eval}(\sigma_{n-1}[\phi](bx)) = \tau
\]
Proof: By (3)1, \(3)5, \text{ and definition of the execution graph of state machines (Def. 11).}

(3)7. \(\text{eval}(\sigma'_2[\phi](bx_2)) = \mathit{t}
\]

(4)1. Choose \((Q_2, V_2) (e', bx_2, as'_2) (Q_3, V_3) \in \mathcal{R}' \text{ such that}
\]
\[\text{eval}(\sigma'_2[\phi](bx_2)) = \tau
\]
PROOF: By (3)3, (3)5, and definition of the execution graph of state machines (Def. 11).

(4.2. \( bx'_2 = \text{conj}(b, \text{neg}(\text{disj}(b_1, \ldots, bx, \ldots, b_k))) \) for some \( b, b_1, \ldots, b_k \in \text{BExp} \)

(5.1. Choose \( Ix \in \mathbb{P}(\mathbb{N}) \) such that \( Q_3 = Q_2 \cup \text{St}(Q_2, e', Ix) \)

PROOF: By (4.1) and definition of \( ph2 \) (Def. 26).

(5.2. Choose

\[
q'_1, \ldots, q'_m \in Q_2 \\
p'_1, \ldots, p'_m \in Q \\
bx_1, \ldots, bx_m \in \text{BExp} \\
as_1, \ldots, as_m \in (\text{Var} \times \text{Exp})^*
\]

such that

\[
 Vi(Q_2, e', Ix, V_2) = \\
 \text{set}(Ix \oplus ((q'_1, (e', bx_1, as_1), p'_1), \ldots, (q'_m, (e', bx_m, as_m), p'_m))
\]

where \( Vi \) is the function defined in Def. 26.

PROOF: By (4.1) and definition of \( Vi \) (Def. 26).

(5.3. Choose \( j \in \mathbb{N} \) such that

\[
(q'_j, (e', bx_j, as_j), p_j) = (q_{n-1}, (e', bx, as), q_n)
\]

PROOF: By (5.2), (3)4 (since \( q_{n-1} \in Q_2 \)), and (3)6 (since \( q_{n-1} \rightarrow q_n \in R \)).

(5.4. \( j \not\in Ix \)

(6.1. \text{ASSUME: } 3.1 \ j \in Ix

PROOF: False

(7.1. \( \text{St}(Q_2, e', Ix, V_2) \cap F \neq \emptyset \) where \( \text{St} \) is the function defined in Def. 26

PROOF: By (5.2), (5.3), assumption 3.1, and definition of \( \text{St} \) (Def. 26), we have that \( p_j \in \text{St}(Q_2, e', Ix, V_2) \). By (5.3) we know that \( p_j = q_n \). Since \( q_n \in F \) (by (3)1), this implies that \( p_j \in F \) which again implies that (7.1) holds.

(7.2. \text{Q.E.D.}

PROOF: (7.1) contradicts (4.1) by definition of \( ph2 \) (Def. 26).

(6.2. \text{Q.E.D.}

By (6.1), \( j \in Ix \) leads to a contradiction, therefore \( j \not\in Ix \).

(5.5. \( bx_j \in \text{set}(\mathbb{N} \setminus Ix \oplus (bx_1, \ldots, bx_m)) \)

PROOF: By (5.2), (5.4) and definition of \( \oplus \) and \( \text{set} \) (see App. D.1.2).

(5.6. \( bx'_2 = \text{conj}(Ix \oplus (bx_1, \ldots, bx_m), \text{neg}(\text{disj}(\mathbb{N} \setminus Ix \oplus (bx_1, \ldots, bx_m)))) \)

PROOF: By (4.1), (5.2) and definition of \( ph2 \) (Def. 26).

(5.7. \text{Q.E.D.}

PROOF: By (5.5) and (5.6).

(4.3. \text{Q.E.D.}

PROOF: By (4.1), (4.2), and definition of \( \text{conj}, \text{disj}, \) and \( \text{neg.} \)

(3.8. \( \text{eval}(\sigma_{n-1}[\emptyset](bx)) = \text{eval}(\sigma'_2[\emptyset](bx)) \)

(4.1. \( \sigma_{n-1} \cap (\text{var}(bx) \times \text{Exp}) = \sigma'_2 \cap (\text{var}(bx) \times \text{Exp}) \)

(5.1. \( \text{[q_1, \sigma_1] \xrightarrow{\sigma_1} [q_0, \sigma_0] \xrightarrow{\sigma_1} [q_1, \sigma_1] \xrightarrow{\sigma_2} [q_2, \sigma_2] \cdots \xrightarrow{\sigma_n} [q_{n-1}, \sigma_{n-1}] \in EG(SM) \)

PROOF: By (3.1).

(5.2. \( \text{[q'_1, \sigma'_1] \xrightarrow{\sigma'_1} ([Q_1, V_1], \sigma'_1] \xrightarrow{\sigma'_2} ([Q_2, V_2], \sigma'_2] \in EG(SM') \)

PROOF: By (3.3).
(5.3) \( q_{n-1} \xrightarrow{(e',bx,as)} q_n \in R \)

Proof: By (3)6.

(5.4) \( \text{comp}([SM], (e_1, e_2, \ldots, e_{n-1}), u) \)

Proof: By (3)2.

(5.5) Q.E.D.

Proof: By (5.1) - (5.4), assumption 2, and definition of side-effect freedom (Def. 25).

(4.2) \( \sigma_{n-1} \phi \cap (\text{var}(bx) \times \text{Exp}) = \sigma_n' \phi \cap (\text{var}(bx) \times \text{Exp}) \)

Proof: By (4)1 and definition of the data state overriding operator

(4.3) Q.E.D.

Proof: By (4)2 and definition of eval (see App. A.2.2).

(3.9) Q.E.D.

Proof: We have that (3)8 contradicts (3)6 (which asserts \( \text{eval}(\sigma_{n-1} \phi)(bx) = \top \)) and (3)7 (which asserts \( \text{eval}(\sigma_n' \phi)(bx) = \bot \)).

(2.2) Q.E.D.

Proof: By contradiction.

(1.3) Q.E.D.

Proof: By (1)1 and (1)2.

**Corollary 1.1** The proof is based on induction. We make use of the predicate 
\( \text{Ind} \in SM \times SM \times \mathbb{N} \rightarrow \mathbb{B} \) which high-lights the induction:

\[
\text{Ind}(\langle E, Q, R, q_1, F \rangle, \langle E', Q', R', q'_1, F' \rangle, m) \equiv \forall n \in \mathbb{N} : \\
\forall q_1 \in Q \setminus F : \cdots : \forall q_n \in Q \setminus F : \\
\forall \sigma_1 \in \Sigma_T : \cdots : \forall \sigma_n \in \Sigma_T : \\
\forall (Q_1, V_1), (Q_n, V_n) \in Q' : \\
\forall \sigma'_1 \in \Sigma_T : \forall \sigma'_n \in \Sigma_T : \\
\forall e_1 \in E : \cdots : \forall e_n \in E : \forall u \in E^* \\
\land n \leq m \\
\land \text{comp}([SM], (e_1, \ldots, e_{n-1}), u) \\
\land [q_1, \sigma_1] \xrightarrow{e_1} [q_1, \sigma_1] \xrightarrow{e_2} [q_2, \sigma_2] \cdots \\
\xrightarrow{e_n} [q_n, \sigma_n] \in EG(\langle E, Q, R, q_1, F \rangle) \\
\land [q_1, \sigma_1] \xrightarrow{e_1} [Q_1, V_1, \sigma_1'] \xrightarrow{e_2} [Q_n, V_n, \sigma_n'] \in EG(\langle E', Q', R', q_1', F' \rangle) \\
\implies q_n \in Q_n
\]

**Assume:** 1. \( SM = \langle E, Q, R, q_1, F \rangle \) for \( SM \in SM \) satisfying \( SM_1 - SM_4 \) and 
\( \langle \rangle \notin [SM] \)
2. \( ph2 \) is side effect free for \( SM \)
3. \( SM' = ph2(SM) = \langle E', Q', R', q'_1, F' \rangle \)
4. \( n \in \mathbb{N} \)
5. \( q_1, \ldots, q_n \in Q \setminus F \)
6. \( \sigma_1, \ldots, \sigma_n \in \Sigma_T \)
7. \( (Q_1, V_1), (Q_n, V_n) \in Q' \)
8. \( \sigma'_1, \sigma'_n \in \Sigma_T \)
9. \( e_1, \ldots, e_n \in E \)
10. \( \text{comp}([SM], \langle e_1, \ldots, e_n \rangle, u) \) for \( u \in E^* \)
11. \( [q_1, \sigma_1] \xrightarrow{e_1} [q_1, \sigma_1] \xrightarrow{e_2} [q_2, \sigma_2] \cdots \xrightarrow{e_n} [q_n, \sigma_n] \in EG(SM) \)
12. \([q'_1, \sigma'_1] \xrightarrow{(Q_1, V_n), \sigma'_1} [(Q_n, V_n), \sigma'_n] \in EG(SM')\)

PROVE: \(q_n \in Q_n\)

(1) Case: 1.1 \(n = 0\)

PROOF: Case assumption 1.1 leads to a contradiction. To see this, note that by case assumption 1.1, assumption 11, and definition of \([\_\_\_]\) (Def. 12) have \(\langle \_ \rangle \in \langle SM \rangle\) which contradicts assumption 1.

(1) Case: 1.1 \(Ind(SM, SM', n - 1)\)

(2) Choose \(u' \in E^*\) such that \(u' \not\subseteq \langle e_n \rangle\) \(\cap comp([SM], \langle e_1, \ldots, e_{n-1} \rangle, u')\)

(4) \(pr([SM], \langle e_1, \ldots, e_{n-1} \rangle) \neq 0\)

PROOF: By assumption 10 and definition of \(pr\) (Def. 25).

(2) Case: \(\langle e_1, \ldots, e_{n-1} \rangle < u'\)

PROOF: By assumption 10 and definition of \(comp([SM], \langle e_1, \ldots, e_{n-1} \rangle)\) (Def. 25).

(4) \(\exists s' \in pr([SM], \langle e_1, \ldots, e_{n-1} \rangle) : \langle e_1, \ldots, e_{n-1} \rangle \cap s' \land s' < u'\)

PROOF: By (4) 2.

(4) Q.E.D.

PROOF: By (4) 1 - (4) 3 and definition of \(comp([SM], \langle e_1, \ldots, e_{n-1} \rangle)\) (Def. 25).

(2) Choose \((Q_{n-2}, V_{n-2}), (Q_{n-1}, V_{n-1}) \in \mathcal{Q}'\) and \(\sigma'_{n-2}, \sigma'_{n-1} \in \Sigma_T\) such that \([q'_1, \sigma'_1] \xrightarrow{\langle Q_1, V_1, \sigma'_1 \rangle} [(Q_{n-2}, V_{n-2}), \sigma'_{n-2}]' \xrightarrow{\langle e_n \rangle} [(Q_{n-1}, V_{n-1}), \sigma'_{n-1}] \in EG(SM')\)

PROOF: By assumption 12, (2) 1 and definition of the execution graph for state machines (Def. 11).

(2) \(q_{n-1} \in Q_{n-2}\)

PROOF: By (2) 1, (2) 2, assumptions 4-12, and induction hypothesis 1.1.

(2) Choose \((e'_n, bx, as) \in \mathcal{A}\) and \(\phi \in \hat{\Sigma}\)

such that

\(q_{n-1} \xrightarrow{\langle e'_n, bx, as \rangle} q_n \in \mathcal{R},\)

\(\text{Dom}(\phi) = var(e'_n),\)

\(\phi(e'_n) = e_n,\) and

\(eval(\sigma_{n-1}[\phi](bx)) = t\)

PROOF: By assumption 11 and definition of the execution graph for state machines (Def. 11).

(2) \(q_n \in Q_{n-1}\)

(3) \(\text{Choose } \mathcal{I}x \in \mathcal{P}(\mathbb{N})\) such that

\(\langle Q_{n-1}, V_{n-1} \rangle = (Q_{n-2} \cup St(Q_{n-2}, e'_n, \mathcal{I}x, V_{n-2}), (V_{n-2} \cup Vi(Q_{n-2}, e'_n, \mathcal{I}x, V_{n-2})) \setminus V i(St(Q_{n-2}, e'_n, \mathcal{I}x, V_{n-2})))\)

where \(St\) is the function defined in Def. 26.

PROOF: By (2) 5 and definition of \(ph2\) (Def. 26).

(3) \(q_n \in Q_{n-1}\) \(\square\)
\[ q'_1, \ldots, q'_m \in Q_{n-2} \]
\[ p'_1, \ldots, p'_m \in Q \]
\[ bx_1, \ldots, bx_m \in BExp \]
\[ as_1, \ldots, as_m \in (\text{Var} \times \text{Exp})^* \]
such that
\[ V_i(Q_{n-2}, e'_n, Ix, V_{n-2}) = \]
\[ \text{set}(Ix \oplus ((q'_1, \langle e'_n, bx_1, as_1 \rangle), \ldots, (q'_m, \langle e'_n, bx_m, as_m \rangle, p'_m))) \]
where \( V_i \) is the function defined in Def. 26.

**Proof:** By definition of \( V_i \) (Def. 26).

\( \langle 3 \rangle \). Choose \( j \in \mathbb{N} \) such that
\[ (q'_j, bx_j, as_j) = (q_{n-1}, \langle e'_n, bx, as \rangle, q_n) \]
**Proof:** By \( \langle 2 \rangle 3 \), \( \langle 2 \rangle 4 \), \( \langle 3 \rangle 2 \) and definition of \( V_i \) (Def. 26).

\( \langle 3 \rangle 4 \). \( j \in Ix \)
\[ \langle 4 \rangle 1. \text{Assume: } 2.1 j \notin Ix \]
**Prove:** False
\[ \langle 5 \rangle 1. \text{eval}(\sigma'_{n-2}[\phi](bx)) = \top \]
\[ \langle 6 \rangle 1. bx' = \text{conj}(Ix \oplus (bx_1, \ldots, bx_m), \text{neg}(\mathbb{N} \setminus Ix \oplus (bx_1, \ldots, bx_m))) \]
**Proof:** By \( \langle 3 \rangle 1 \), \( \langle 3 \rangle 2 \), and definition of \( ph2 \) (Def. 26).
\[ \langle 6 \rangle 2. bx_j \in \text{set}(\mathbb{N} \setminus Ix \oplus (bx_1, \ldots, bx_m)) \]
**Proof:** By assumption 2.1 and definition of \( \oplus \) (see App. D.1.2).

\( \langle 3 \rangle 3 \). Q.E.D.

**Proof:** By \( \langle 6 \rangle 1 \), and \( \langle 6 \rangle 2 \), definition of \( \text{conj}, \text{disj} \), and \( \text{neg} \) (see App. D.1.2), \( bx \) must evaluate to false (i.e., \( \text{eval}(\sigma'_{n-2}[\phi](bx)) = \top \)) since \( bx' \) evaluates to true (\( \text{eval}(\sigma_{n-1}[\phi](bx)) = \bot \)) by \( \langle 2 \rangle 5 \).

\[ \langle 5 \rangle 2. \text{eval}(\sigma_{n-1}[\phi](bx)) = \text{eval}(\sigma'_{n-2}[\phi](bx)) \]
\[ \langle 6 \rangle 1. [q_1, \sigma_1] \xrightarrow{\varepsilon_1} [q_1, \sigma_1] \xrightarrow{\varepsilon_2} \cdots \xrightarrow{\varepsilon_{n-1}} [q_{n-1}, \sigma_{n-1}] \in \text{EG}(SM) \]
**Proof:** By assumption 13.
\[ \langle 6 \rangle 2. [q_1, \sigma_1] \xrightarrow{\varepsilon_1} [(Q_1, V_1), \sigma'_1] \xrightarrow{\varepsilon'} [(Q_{n-2}, V_{n-2}), \sigma'_{n-2}] \in \text{EG}(SM') \]
**Proof:** By assumption 14.
\[ \langle 6 \rangle 3. q_{n-1} \xrightarrow{(e'_n, bx, as)} q_n \in \mathcal{R} \]
**Proof:** By \( \langle 2 \rangle 4 \).
\[ \langle 6 \rangle 4. \text{comp}([SM], \{e_1, \ldots, e_{n-1}\}, v') \]
**Proof:** By \( \langle 2 \rangle 1 \).

\[ \langle 6 \rangle 5. \text{Q.E.D.} \]
**Proof:** By \( \langle 6 \rangle 1 \) - \( \langle 6 \rangle 4 \), assumption 2, and definition of side effect freedom (Def. 25).

\( \langle 5 \rangle 3. \text{Q.E.D.} \]
**Proof:** \( \langle 5 \rangle 1 \) and \( \langle 5 \rangle 2 \) contradict \( \langle 2 \rangle 4 \) (since it asserts that \( \text{eval}(\sigma_{n-1}[\phi](bx)) = \top \))

\( \langle 4 \rangle 2. \text{Q.E.D.} \]
**Proof:** By contradiction.

\( \langle 3 \rangle 5. q_n \in St(Q_{n-2}, e'_n, Ix, V_{n-2}) \) where \( St \) is the function defined in Def. 26.
**Proof:** By \( \langle 3 \rangle 2 \) - \( \langle 3 \rangle 4 \), and definition of \( St \) (Def. 26).

\( \langle 3 \rangle 6. \text{Q.E.D.} \]
**Proof:** By \( \langle 3 \rangle 1 \) and \( \langle 3 \rangle 5 \).

\( \langle 2 \rangle 7. \text{Q.E.D.} \]
**Proof:** By \( \langle 2 \rangle 6 \) since by definition of \( ph2 \) (Def. 26), we know that \( (Q, V) \xrightarrow{\varepsilon} (Q', V') \in \text{EG}(SM') \) implies \( Q \subseteq Q' \).
Corollary 1.2  The proof is based on induction. We make use of the predicate $\text{Ind} \in \text{SM} \times \text{SM} \times E^* \rightarrow B$ which high-lights the induction. Let $SM = \omega^{\epsilon_1} (E, Q, R, q_1, F)$, and $SM' = (E', Q', R', q'_1, F')$, then $\text{Ind}$ is defined by

$$\text{Ind}(SM, SM', s) \overset{\text{def}}{=} \forall s' \in E^* :$$

$$\forall (Q_n, V_n) \in Q' : \forall q_n \in Q \cap Q_n :$$

$$\forall \sigma_1 \in \Sigma_T : \forall \sigma'_n \in \Sigma_T :$$

$$\wedge s' \subseteq s$$

$$\wedge [q'_1, \sigma_1] \overset{(\epsilon)\rightarrow s'}{\rightarrow} [(Q_n, V_n), \sigma'_n] \in \text{EG}(SM')$$

$$\exists \sigma_n \in \Sigma_T :$$

$$\wedge [q_1, \sigma_1] \overset{(\epsilon)\rightarrow}{\rightarrow} [q_n, \sigma_n] \in \text{EG}(SM)$$

$$\wedge \text{comp}([SM], u, s')$$

Assume: 1. $SM = (E, Q, R, q_1, F)$ for $SM \in \text{SM}$ satisfying $\text{SM1} - \text{SM4}$ and

$\langle \rangle \notin [SM]$  
2. $ph2$ is side effect free for $SM$  
3. $SM' = ph2(SM) = (E', Q', R', q'_1, F')$  
4. $s \in E^*$  
5. $(Q_n, V_n) \in Q'$  
6. $\sigma_1, \sigma'_n \in \Sigma_T$  
7. $q_n \in Q_n \cap Q$  
8. $[q'_1, \sigma_1] \overset{(\epsilon)\rightarrow s}{\rightarrow} [(Q_n, V_n), \sigma'_n] \in \text{EG}(SM')$

Prove: $\exists u \in E^*$  

$$\exists \sigma_n \in \Sigma_T :$$

$$\wedge [q_1, \sigma_1] \overset{(\epsilon)\rightarrow}{\rightarrow} [q_n, \sigma_n] \in \text{EG}(SM)$$

$$\wedge \text{comp}([SM], u, s')$$

(1) 1. Case: $1.1 \ s = \langle \rangle$

(2) 1. Choose $q_1 \overset{(\epsilon, \epsilon, \text{osa})}{\rightarrow} q' \in R$  

Proof: By assumption 1 since $SM$ satisfies syntax constraint $\text{SM2}$.  
(2) 2. $q'_1 \overset{(\epsilon, \epsilon, \text{osa})}{\rightarrow} \{q'\} \in R'$

Proof: By (2) 1, assumption 3, and definition of $ph2$ (Def. 26).  
(2) 3. $[q'_1, \sigma_1] \overset{(\epsilon)\rightarrow}{\rightarrow} [(Q_n, V_n), \sigma'_n] \in \text{EG}(SM')$

Proof: By assumption 8, case assumption 1.1, and assumptions 1 (since $SM$ is assumed to satisfy syntax constraint $\text{SM3}$) and 3 and definition of $ph2$ (Def. 26).  
(2) 4. $q_n = q'$

Proof: By assumption 7, (2) 2, and (2) 3 and definition of $ph2$ (Def. 26) (since $Q_n = \{q'\}$).  
(2) 5. Choose $\sigma_n \in \Sigma_T$ such that $[q_1, \sigma'_1] \overset{(\epsilon)\rightarrow}{\rightarrow} [q_n, \sigma_n] \in \text{EG}(SM)$

Proof: By (2) 1, (2) 4 and definition of the execution graph for state machines (Def. 11).  
(2) 6. Q.E.D.
Proof: By (2) 5.

(1) 2. Case: 1.1 $s = s' \sim \langle e \rangle$

1.2 Ind($SM, SM', s'$)

(2) 1. Choose $(Q_{n-1}, V_{n-1}) \in Q'$ and $\sigma'_{n-1} \in \tilde{\Sigma}_T$ such that $[q'_l, \sigma'_l] \xrightarrow{(c)\cdot s'} [(Q_{n-1}, V_{n-1}), \sigma'_{n-1}] \xrightarrow{\sigma'_{n-1}} [(Q_{n}, V_{n}), \sigma'_{n}] \in EG(SM')$

Proof: By assumption 8, assumption 1.1, and definition of the execution graph for state machines (Def. 11).

(2) 2. Choose $(e', bx', as') \in \text{Act}$ and

\[ \phi \in \tilde{\Sigma} \]

such that

$(Q_{n-1}, V_{n-1}) \xrightarrow{(e', bx', as')} (Q_{n}, V_{n}) \in R'$

$\text{Dom}(\phi) = \text{var}(e')$, $\phi(e') = e$, and $\text{eval}(\sigma'_{n-1}[\phi](bx')) = t$

Proof: By (2) 1 and definition of the execution graph for state machines (Def. 11).

(2) 3. Case: 2.1 $\exists Ix \in \mathbb{P}(\mathbb{N}) : q_n \in St(Q_{n-1}, e', Ix, V_{n-1})$

(3) 1. Choose $Ix \in \mathbb{P}(\mathbb{N})$ and $q_{n-1} \xrightarrow{(e', bx, sa)} q_n \in R$ such that $q_{n-1} \xrightarrow{(e', bx, sa)} q_n \in V(i(Q_{n-1}, e', Ix, V_{n-1})$

Proof: By case assumption 2.1 and definition of $Vi$ (Def. 26).

(3) 2. Choose $\sigma_{n-1} \in \tilde{\Sigma}_T$ and,

$u' \in \mathbb{E}^*$

such that

(A) $[q_l, \sigma_l] \xrightarrow{(c)\cdot u'} [q_{n-1}, \sigma_{n-1}] \in EG(SM)$,

(B) $\text{comp}([\tilde{\Sigma} M], u', s')$

Proof: By assumptions 1.1, 1.2, and (3) 1.

(3) 3. Choose $\sigma_n \in \tilde{\Sigma}_T$ such that

$[q_{n-1}, \sigma_{n-1}] \xrightarrow{(c)\cdot u'} [q_n, \sigma_n] \in EG(SM)$

Proof: By (2) 1, (2) 2, (3) 2, assumption 2, and assumption 3.

(3) 4. $\text{comp}([\tilde{\Sigma} M], u' \sim \langle e \rangle, s' \sim \langle e \rangle)$

(4) 1. $\text{pr}([\tilde{\Sigma} M], u' \sim \langle e \rangle) \neq \emptyset$

Proof: By (3) 2 and definition of $\text{pr}$ (Def. 25).

(4) 2. $u' \sim \langle e \rangle \sim s' \sim \langle e \rangle$

Proof: By (3) 2 and definition of $\text{comp}([\tilde{\Sigma} M]$ (Def. 25).

(4) 3. $\sim (\exists u'' \in \text{pr}([\tilde{\Sigma} M], u' \sim \langle e \rangle)) : u' \sim \langle e \rangle \in s'' \wedge s'' \sim u' \sim \langle e \rangle)$

Proof: If (4) 3 does not holds, then we can choose $e' \in \mathbb{E}$ such that $u' \sim \langle e', e' \rangle \sim s' \sim \langle e \rangle$, but by definition of $\sim$, this is impossible.

(4) 4. Q.E.D.

Proof: By (4) 1 - (4) 3 and definition of $\text{comp}([\tilde{\Sigma} M]$ (Def. 25).

(3) 5. Q.E.D.

Proof: By (3) 2 - (3) 4.

(2) 4. Case: 2.1 $\neg (\exists Ix \in \mathbb{P}(\mathbb{N}) : q_n \in St(Q_{n-1}, e', Ix, V_{n-1}))$

(3) 1. Choose $q_{n-1} \in Q_{n-1} \cap Q$,

$\sigma'_{n-1} \in \tilde{\Sigma}_T$ and,

$u' \in \mathbb{E}^*$
such that
\[(A) [q_I, \sigma_I] \xrightarrow{[c]} [q_{n-1}, \sigma_{n-1}] \in EG(SM),
(B) \text{comp}(\llbracket SM \rrbracket, u', s'),
(C) q_{n-1} = q_n\]

PROOF: (A) and (B) hold by case assumption 1.2. (C) holds by case assumption 7, assumption 2.1, and definition of \(ph2\) (Def. 26) since we have \(Q_n = Q_{n-1}\).

\[\langle 3.2. [q_{n-1}, \sigma_{n-1}] \xrightarrow{[c]} [q_n, \sigma_{n-1}] \in EG(SM) \rangle\]

PROOF: By (3)1 and definition of the execution graph for state machines (Def. 11).

\[\langle 3.3. \text{comp}(\llbracket SM \rrbracket, u', s' \sim (e)) \rangle\]

\[\langle 4.1. \text{pr}(\llbracket SM \rrbracket, u') \neq 0 \rangle\]

PROOF: By (3)1 and definition of \(pr\) (Def. 25).

\[\langle 4.2. u' \sim s' \sim (e) \rangle\]

PROOF: By (3)1 and definition of \(\text{comp}(\_ \_ \_ \_ \_ \_ \_ \_)\) (Def. 25).

\[\langle 4.3. \neg \exists s'' \in \text{pr}(\llbracket SM \rrbracket, u') : u' \sim (e) \sqsubseteq s'' \wedge s'' \sim u' \sim (e) \rangle\]

PROOF: If (4)3 holds, then there must be trace \(t \in \llbracket SM \rrbracket\) such that \(u' \sim (e) \sqsubseteq t\). However, this contradicts case assumption 2.1.

\[\langle 4.4. \text{Q.E.D.} \rangle\]

PROOF: By (4)1 - (4)3 and definition of \(\text{comp}(\_ \_ \_ \_ \_ \_ \_ \_)\) (Def. 25).

\[\langle 3.4. \text{Q.E.D.} \rangle\]

PROOF: By (3)1 - (3)3.

\[\langle 2.5. \text{Q.E.D.} \rangle\]

PROOF: By (2)1 - (2)4.

\[\langle 1.3. \text{Q.E.D.} \rangle\]

PROOF: By (1)1 and (1)2 and induction.

**Theorem 1** Let \(d\) be a well formed single lifeline sequence diagram such that \(ph2\) is side effect free for \(ph1(d)\), then the transformation \(d2p(d)\) is adherence preserving, i.e.,

\[d \rightarrow_{da} \Phi \Leftrightarrow d2p(d) \rightarrow_{sa} \Phi \quad \text{for all systems } \Phi\]

**Proof of Theorem 1**

**ASSUME:** 1. \(d \in D'\) for some \(l \in L\)

2. \(d\) satisfies conditions \(SD1 - SD10\)

3. \(ph2\) is side effect free for \(ph1(d)\)

**PROVE:** \(d \rightarrow_{da} \Phi \Leftrightarrow d2p(d) \rightarrow_{sa} \Phi\)

\[\langle 1.1. \text{ASSUME: } 1.1 \ d \rightarrow_{da} \Phi \rangle\]

**PROVE:** \(d2p(d) \rightarrow_{sa} \Phi\)

\[\langle 2.1. \Phi|_{E^{l}} \subseteq [d2p(d)] \rangle\]

\[\langle 3.1. \text{ASSUME: } 2.1 \ t \in \Phi|_{E^{l}} \rangle\]

**PROVE:** \(t \in [d2p(d)]\)

\[\langle 4.1. \forall s \in [ph1(d)] : \neg (s \sim t) \rangle\]

**PROOF:** By assumptions 1, 2, 1.1, and 2.1, Lemma 1, and definition of \(\rightarrow_{da}\) (Def 8).

\[\langle 4.2. \ t \in [ph2(ph1(d))] \rangle\]

**PROOF:** By assumption 2.1, \(\langle 4.1 \rangle\), assumptions 1-3, and Corollary 1.

\[\langle 4.3. \text{Q.E.D.} \rangle\]
Proof: By (4) 2 and definition of $d2p$ (Def. 18).

(3) 2. Q.E.D.

Proof: By (3) 1 and definition of $\subseteq$.

(2) 2. Q.E.D.

Proof: By (2) 1 and definition of $\rightarrow_{sa}$ (Def. 13).

(1) 2. Assume: 1.1 $d2p(d) \rightarrow_{sa} \Phi$

Prove: $d \rightarrow_{da} \Phi$

(2) 1. Assume: 2.1 $s \in H_{neg}$ for $[d] = (H_{pos}, H_{neg})$

$2.2 \ t \in \Phi|_d$

Prove: $\neg(s < t)$

(3) 1. $t \in \llbracket ph2(ph1(d)) \rrbracket$

Prove: By assumption 1.1, assumption 2.2, and definition of $\rightarrow_{sa}$ (Def. 13).

(3) 2. $s \in \llbracket ph1(d) \rrbracket$

Prove: Assumptions 1 and 2, assumption 2.1, and Lemma 1.

(3) 3. Q.E.D.

Proof: By (3) 1, (3) 2, assumptions 1-3, and Corollary 1.

(2) 2. Q.E.D.

Proof: By (2) 1 and definition of $\rightarrow_{da}$ (Def. 8).

(1) 3. Q.E.D.

Proof: By (1) 1 and (1) 2.

Theorem 2 Let $d$ be a well formed sequence diagram such that $ph2$ is side effect free for $ph1(\pi_l(d))$ for all lifelines $l$ in $d$, then the transformation $d2p(d)$ is adherence preserving, i.e.,

$$d \rightarrow_{dag} \Phi \iff d2p(d) \rightarrow_{sag} \Phi$$

for all systems $\Phi$.

Proof of Theorem 2

Assume: 1. $d \in D$ and $d$ satisfies conditions SD1 - SD10

2. $ph2$ is side effect free for $ph1(\pi_l(d))$ for all $l \in ll.d$

Prove: $d \rightarrow_{dag} \Phi \iff d2p(d) \rightarrow_{sag} \Phi$

(1) 1. Assume: 1.1 $d \rightarrow_{dag} \Phi$

Prove: $d2p(d) \rightarrow_{sag} \Phi$

(2) 1. $d2p(\pi_l(d)) \rightarrow_{sa} \Phi$ for all $l \in ll.d$

(3) 1. $\pi_l(d) \rightarrow_{da} \Phi$ for all $l \in ll.d$

Proof: By assumption 1.1 and definition of $\rightarrow_{dag}$ (Def. 9)).

(3) 2. Q.E.D.

Proof: By (3) 1, assumptions 1 and 2, and Theorem 1.

(2) 2. Q.E.D.

Proof: By (2) 1, definition of $\rightarrow_{sag}$ (Def. 14)), and definition of $d2p$ (Def. 28).

(1) 2. Assume: 1.1 $d2p(d) \rightarrow_{sag} \Phi$

Prove: $d \rightarrow_{dag} \Phi$

(2) 1. $\pi_l(d) \rightarrow_{da} \Phi$ for all $l \in ll.d$

(3) 1. $d2p(\pi_l(d)) \rightarrow_{sa} \Phi$ for all $l \in ll.d$

Proof: By assumption 1.1 and definition of $\rightarrow_{sag}$ (Def. 14)), and definition of $d2p$ (Def. 28).

(3) 2. Q.E.D.
Proof: By (3)1, assumptions 1 and 2, and Theorem 1.

(2)2. Q.E.D.

Proof: By (2)1 and definition of $\rightarrow_{\text{dag}}$ (Def. 9).

(1)3. Q.E.D.

Proof: By (1)1 and (1)2.